Monads for natural language semantics

Chung-chieh Shan
Computer Science
Harvard University
ken@digitas.harvard.edu

Abstract

Accounts of semantic phenomena often involve extending types of meanings and revising composition rules at the same time. The concept of *monads* allows many such accounts—for intensionality, variable binding, quantification and focus—to be stated uniformly and compositionally.

1 Introduction

The Montague grammar tradition formulates formal semantics for natural languages in terms of the λ-calculus. Each utterance is considered a tree in which each leaf node is a lexical item whose meaning is a (usually typed) value in the λ-calculus. The leaf node meanings then determine meanings for subtrees, through recursive application of one or more *composition rules*. A composition rule specifies the meaning of a tree in terms of the meanings of its sub-trees. One simple composition rule is function application:

$$[\![x\,y]\!] = [\![x]\!]\big([\![y]\!]\big) : \beta \qquad \text{where } [\![x]\!] : \alpha \to \beta \text{ and } [\![y]\!] : \alpha. \tag{1}$$

Here α and β are type variables, and we denote function types by $\to$.

To handle phenomena such as intensionality, variable binding, quantification and focus, we often introduce new types in which to embed existing aspects of meaning and accommodate additional ones. Having introduced new types, we then need to revise our composition rules to reimplement existing functionality. In this way, we often augment semantic theories by simultaneously extending the types of meanings and stipulating new composition rules. When we augment a grammar, its original lexical meanings and composition rules become invalid and require global renovation (typically described as "generalizing to the worst case" (Partee 1996)). Each time we consider a new aspect of meaning, all lexical meanings and composition rules have to be revised.

Over the past decade, the category-theoretic concept of *monads* has gained popularity in computer science as a tool to structure denotational semantics (Moggi 1990, 1991) and functional programs (Wadler 1992a,b) with computational side effects. When used to structure computer programs, monads allow the substance of a computation to be defined separately from the plumbing that supports its execution, increasing modularity. Many accounts of phenomena in natural language semantics can also be phrased in terms of monads, thus clarifying the account and simplifying the presentation.

In this paper, I will present the concept of monads and show how they can be applied to natural language semantics. To illustrate the approach, I will use four monads to state analyses of well-known phenomena *uniformly* and *compositionally*. By "uniformly" I mean that, even though the analyses make use of a variety of monads, they all invoke monad primitives in the same way. By "compositionally" I mean that the analyses define composition rules in the spirit of Montague grammar. After presenting the monadic analyses, I will discuss combining monads to account for interaction between semantic phenomena.

Acknowledgments Thanks to Stuart Shieber, Dylan Thurston, Chris Barker, and the anonymous referees at ESSLLI-2001 and WEB-SLS for helpful discussions and comments. This work is supported by the United States National Science Foundation under Grant IRI-9712068.

2 Monadic analyses

Intuitively, a monad is a transformation on types equipped with a composition method for transformed values. Formally, a monad is a triple $(\mathbb{M}, \eta, \star)$, where $\mathbb{M}$ is a type constructor (a map from each type α to a corresponding type $\mathbb{M}\alpha$), and η and $\star$ are functions (pronounced "unit" and "bind")

$$\eta : \alpha \to \mathbb{M}\alpha, \qquad \star : \mathbb{M}\alpha \to (\alpha \to \mathbb{M}\beta) \to \mathbb{M}\beta. \tag{2}$$

These two functions are polymorphic in the sense that they must be defined for all types α and β. Roughly speaking, η specifies how ordinary values can be injected into the monad, and $\star$ specifies how computations within the monad compose with each other. By definition, η and $\star$ must satisfy left identity, right identity, and associativity:

$$\eta(a) \star k = k(a) \qquad \forall a : \alpha,\ k : \alpha \to \mathbb{M}\beta, \tag{3a}$$

$$m \star \eta = m \qquad \forall m : \mathbb{M}\alpha, \tag{3b}$$

$$\big(m \star k\big) \star l = m \star \big(\lambda v.\, k(v) \star l\big) \qquad \forall m : \mathbb{M}\alpha,\ k : \alpha \to \mathbb{M}\beta,\ l : \beta \to \mathbb{M}\gamma. \tag{3c}$$

In computer science, monads formalize the idea that denotations are not values but value-producing *computations*. For every value type α, the monad specifies a type $\mathbb{M}\alpha$, the type of computations that produce α-values (possibly with some side effect). The function η maps every α-value a to the trivial computation that produces a with no side effect. Given any α-computation m, and any map k from α-values to β-computations, the β-computation $m \star k$ is the following: First, execute m to compute an α-value a. Then, execute the β-computation $k(a)$.

We now present some concrete linguistic applications of this definition.

2.1 The powerset monad; interrogatives

As a first example, consider sets. Corresponding to each type α we have a type $\alpha \to t$, the type of subsets of α. We define[1]

$$\begin{aligned} \mathbb{M}\alpha &= \alpha \to t && \forall \alpha, && (4a)\\ \eta(a) &= \{a\} : \mathbb{M}\alpha && \forall a : \alpha, && (4b)\\ m \star k &= \textstyle\bigcup_{a \in m} k(a) : \mathbb{M}\beta && \forall m : \alpha \to t,\ k : \alpha \to \beta \to t. && (4c)\end{aligned}$$

The powerset monad is a crude model of non-determinism. For example, the set of individuals m_1 defined by

$$m_1 = \{\text{John}, \text{Mary}\} : \mathbb{M}e$$

can be thought of as a non-deterministic individual—it is ambiguous between John and Mary. Similarly, the function k_1 defined by

$$\begin{aligned} k_1 &: e \to \mathbb{M}(e \to t)\\ k_1(a) &= \{\lambda x.\, \text{like}(a, x), \lambda x.\, \text{hate}(a, x)\} : \mathbb{M}(e \to t)\end{aligned}$$

maps each individual to a non-deterministic property. To apply the function k_1 to the individual m_1, we compute

$$\begin{aligned} m_1 \star k_1 &= \textstyle\bigcup_{a \in \{\text{John},\text{Mary}\}} \{\lambda x.\, \text{like}(a, x), \lambda x.\, \text{hate}(a, x)\}\\ &= \{\lambda x.\, \text{like}(\text{John}, x), \lambda x.\, \text{hate}(\text{John}, x),\\ &\qquad \lambda x.\, \text{like}(\text{Mary}, x), \lambda x.\, \text{hate}(\text{Mary}, x)\} : \mathbb{M}(e \to t).\end{aligned}$$

We see that the non-determinism in both m_1 and k_1 is carried through to produce a 4-way-ambiguous result.

Most words in natural language are not ambiguous in the way m_1 and k_1 are. To upgrade an ordinary (deterministic) value of any type α to the corresponding non-deterministic type $\mathbb{M}\alpha$, we can apply η to the ordinary value, say John:

$$\begin{aligned} \eta(\text{John}) &= \{\text{John}\} : \mathbb{M}e,\\ \{\text{John}\} \star k_1 &= \{\lambda x.\, \text{like}(\text{John}, x), \lambda x.\, \text{hate}(\text{John}, x)\} : \mathbb{M}(e \to t).\end{aligned} \qquad (5)$$

[1]In this section and the next, we treat types as sets in order to define the powerset and pointed powerset monads. These two monads do not exist in every model of the λ-calculus.

Similarly, to convert an ordinary function to a non-deterministic function, we can apply η to the output of the ordinary function, say k_2 below:

$$\begin{aligned} k_2 &= \lambda a.\, \lambda x.\, \mathrm{like}(a,x) : e \to e \to t, \\ \eta \circ k_2 &= \lambda a.\, \{\lambda x.\, \mathrm{like}(a,x)\} : e \to \mathbb{M}(e \to t), \\ m_1 \star (\eta \circ k_2) &= \{\lambda x.\, \mathrm{like}(\mathrm{John},x), \lambda x.\, \mathrm{like}(\mathrm{Mary},x)\} : \mathbb{M}(e \to t). \end{aligned} \tag{6}$$

In both (5) and (6), an ordinary value is made to work with a non-deterministic value by upgrading it to the non-deterministic type.

Consider now the function application rule (1). We can regard it as a two-argument function, denoted A and defined by

$$\begin{aligned} A &: (\alpha \to \beta) \to \alpha \to \beta, \\ A(f)(x) &= f(x) : \beta \qquad \forall f : \alpha \to \beta,\ x : \alpha. \end{aligned} \tag{7}$$

We can lift ordinary function application A to non-deterministic function application $A_{\mathbb{M}}$, defined by

$$\begin{aligned} A_{\mathbb{M}} &: \mathbb{M}(\alpha \to \beta) \to \mathbb{M}\alpha \to \mathbb{M}\beta, \\ A_{\mathbb{M}}(f)(x) &= f \star \big[\lambda a.\, x \star [\lambda b.\, \eta(a(b))]\big] : \mathbb{M}\beta \qquad \forall f : \mathbb{M}(\alpha \to \beta),\ x : \mathbb{M}\alpha. \end{aligned} \tag{8}$$

Substituting (4) into (8), we get

$$A_{\mathbb{M}}(f)(x) = \{\, a(b) \mid a \in f,\ b \in x \,\} \subseteq \beta \qquad \forall f \subseteq \alpha \to \beta,\ x \subseteq \alpha. \tag{9}$$

Just as the definition of A in (7) gives rise to the original composition rule (1), that is,

$$[\![x\, y]\!] = A([\![x]\!])([\![y]\!]), \tag{10}$$

the definition of $A_{\mathbb{M}}$ in (8) gives rise to the revised composition rule

$$[\![x\, y]\!] = A_{\mathbb{M}}([\![x]\!])([\![y]\!]). \tag{11}$$

For the powerset monad, this revised rule is the set-tolerant composition rule in the alternative semantics analysis of interrogatives first proposed by Hamblin (1973). In Hamblin's analysis, the meaning of each interrogative constituent is a set of alternatives available as answers to the question; this corresponds to the definition of $\mathbb{M}$ in (4). By contrast, the meaning of each non-interrogative constituent is a singleton set; this corresponds to the definition of η in (4).

To support question-taking verbs (such as *know* and *ask*), we (and Hamblin) need a secondary composition rule in which A is lifted with respect to the function f but not the argument x:

$$\begin{aligned} [\![x\, y]\!] &= A'_{\mathbb{M}}([\![x]\!])([\![y]\!]) \qquad \text{where} \\ A'_{\mathbb{M}} &: \mathbb{M}(\mathbb{M}\alpha \to \beta) \to \mathbb{M}\alpha \to \mathbb{M}\beta, \\ A'_{\mathbb{M}}(f)(x) &= f \star \big[\lambda a.\, \eta(a(x))\big] : \mathbb{M}\beta \qquad \forall f : \mathbb{M}(\mathbb{M}\alpha \to \beta),\ x : \mathbb{M}\alpha. \end{aligned} \tag{12}$$

Substituting (4) into (12), we get

$$A'_{\mathbb{M}}(f)(x) = \{\, a(x) \mid a \in f \,\} \subseteq \beta \qquad \forall f \subseteq (\alpha \to t) \to \beta,\, x \subseteq \alpha, \tag{13}$$

Note that, for any given pair of types of $[\![x]\!]$ and $[\![y]\!]$, at most one of $A_{\mathbb{M}}$ (8) and $A'_{\mathbb{M}}$ (12) can apply. Thus the primary composition rule (11) and the secondary composition rule (12) never conflict.

2.2 The pointed powerset monad; focus

A variation on the powerset monad (4) is the *pointed powerset monad*; it is implicitly involved in Rooth's (1985; 1996) account of focus in natural language. A *pointed set* is a nonempty set with a distinguished member. In other words, a pointed set x is a pair $x = (x_0, x_1)$, where x_0 is a member of the set x_1. Define the pointed powerset monad by

$$\mathbb{M}\alpha = \bigl\{\, (x_0, x_1) \mid x_0 \in x_1 \subseteq \alpha \,\bigr\} \qquad \forall \alpha, \tag{14a}$$

$$\eta(a) = \bigl(a, \{a\}\bigr) : \mathbb{M}\alpha \qquad \forall a : \alpha, \tag{14b}$$

$$m \star k = \bigl([k(m_0)]_0, \textstyle\bigcup_{a \in m_1} [k(a)]_1\bigr) : \mathbb{M}\beta \qquad \forall m : \mathbb{M}\alpha,\ k : \alpha \to \mathbb{M}\beta. \tag{14c}$$

This definition captures the intuition that we want to keep track of both a set of non-deterministic alternatives and a particular alternative in the set. As with the powerset monad, the definition of $m \star k$ carries the non-determinism in both m and k through to the result.

Substituting our new monad definition (14) into the previously lifted application formula (8) gives

$$A_{\mathbb{M}}(f_0, f_1)(x_0, x_1) = \bigl(f_0(x_0), \{\, a(b) \mid a \in f_1,\ b \in x_1 \,\}\bigr) : \mathbb{M}\beta \\ \forall (f_0, f_1) : \mathbb{M}(\alpha \to \beta),\ (x_0, x_1) : \mathbb{M}\alpha. \tag{15}$$

This formula, in conjunction with the primary composition rule (11), is equivalent to Rooth's recursive definition of focus semantic values.

Crucially, even though the pointed powerset monad extends our meaning types to accommodate focus information, neither our definition of $A_{\mathbb{M}}$ in (8) nor our composition rule (11) needs to change from before. Moreover, the majority of our lexical meanings have nothing to do with focus and thus need not change either. For example, in the hypothetical lexicon entry $[\![\textit{John}]\!] = \eta(\text{John})$, the upgrade from meaning type e to meaning type $\mathbb{M}e$ occurs automatically due to the redefinition of η.

2.3 The reader monad; intensionality and variable binding

Another monad often seen in computer science is the *reader monad*, also known as the *environment monad*. This monad encodes dependence of values on some given input (from the keyboard, say). To define the reader monad, fix a type ρ—say the type s of possible worlds, or the type g of variable assignments—then let

$$\begin{aligned} \mathbb{M}\alpha &= \rho \to \alpha && \forall \alpha, && (16a)\\ \eta(a) &= \lambda w.\, a : \mathbb{M}\alpha && \forall a : \alpha, && (16b)\\ m \star k &= \lambda w.\, k\big(m(w)\big)(w) : \mathbb{M}\beta && \forall m : \mathbb{M}\alpha,\ k : \alpha \to \mathbb{M}\beta. && (16c) \end{aligned}$$

Note how the definition of $m \star k$ threads the input w through both m and k to produce the result. To see this threading process in action, let us once again substitute our monad definition (16) into the definition of $A_{\mathbb{M}}$ in (8):

$$A_{\mathbb{M}}(f)(x) = \lambda w.\, f(w)\big(x(w)\big) : \mathbb{M}\beta \qquad \forall f : \mathbb{M}(\alpha \to \beta),\ x : \mathbb{M}\alpha. \tag{17}$$

For $\rho = s$, we can think of $\mathbb{M}$ as the intensionality monad, noting that (17) is exactly the usual extensional composition rule. While words such as *student* and *know* have meanings that depend on the possible world w, words such as *is* and *and* do not. We can upgrade the latter by applying η.

For $\rho = g$, we can think of $\mathbb{M}$ as the variable binding monad, noting that (17) is the usual assignment-preserving composition rule. Except for pronominals, most word meanings do not refer to the variable assignment. Thus we can upgrade the majority of word meanings by applying η.

Substituting the same monad definition (16) into the secondary composition rule (12) gives

$$A'_{\mathbb{M}}(f)(x) = \lambda w.\, f(w)(x) : \mathbb{M}\beta \qquad \forall f : \mathbb{M}(\mathbb{M}\alpha \to \beta),\ x : \mathbb{M}\alpha. \tag{18}$$

For $\rho = s$, this is the intensional composition rule; it handles sentence-taking verbs such as *know* and *believe* (of type $s \to (s \to t) \to e \to t$) by allowing them to take arguments of type $s \to t$ rather than type t. The monad laws, by the way, guarantee that $A'_{\mathbb{M}}(f)(x) = A_{\mathbb{M}}(f)\big(\eta(x)\big)$ for all f and x; the function η (in this case a map from $s \to t$ to $s \to s \to t$) is simply the intension (up) operator, usually written $^\wedge$.

For $\rho = g$, the same formula (18) is often involved in accounts of quantification that assume quantifier raising at LF, such as that in the textbook by Heim and Kratzer (1998). It handles raised quantifiers (of type $g \to (g \to t) \to t$) by allowing them to take arguments of type $g \to t$ rather than type t. The function η (in this case a map from $g \to t$ to $g \to g \to t$) is simply the variable abstraction operator.

2.4 The continuation monad; quantification

Barker (2001) proposed an analysis of quantification in terms of *continuations*. He *continuizes* a grammar by replacing each meaning type α with its corresponding continuized type $(\alpha \to t) \to t$. As a special case, the meaning type of NPs is changed from e to $(e \to t) \to t$, matching the treatment of English quantification by Montague (1974).

Intuitively, "the continuation represents an entire (default) future for the computation" (Kelsey, Clinger, Rees et al. 1998): In the sentence *John smokes*, the continuation of *John* is the "sentence with a hole" *[–] smokes*. In Barker's account, quantificational NPs such as *everyone* manipulate continuations, for instance to give the meaning of *everyone smokes*.

For any fixed type ω (say t), we can define a *continuation monad* with *answer type* ω:

$$\mathbb{M}\alpha = (\alpha \to \omega) \to \omega \qquad \forall \alpha, \tag{19a}$$

$$\eta(a) = \lambda c.\, c(a) : \mathbb{M}\alpha \qquad \forall a : \alpha, \tag{19b}$$

$$m \star k = \lambda c.\, m\big(\lambda a.\, k(a)(c)\big) : \mathbb{M}\beta \qquad \forall m : \mathbb{M}\alpha,\ k : \alpha \to \mathbb{M}\beta. \tag{19c}$$

The value c manipulated in these definitions is a *continuation*. Each value of type $\mathbb{M}\alpha$ must turn a continuation (of type $\alpha \to \omega$) into an answer (of type ω). The most obvious way to do so, encoded in the definition of η above, is to feed the continuation a value of type α:

$$[\![\textit{John}]\!] = \eta(\text{John}) = \lambda c.\, c(\text{John}) : \mathbb{M}e, \tag{20a}$$

$$[\![\textit{smokes}]\!] = \eta(\text{smoke}) = \lambda c.\, c(\text{smoke}) : \mathbb{M}(e \to t). \tag{20b}$$

To compute the meaning of *John smokes*, we first substitute our monad definition (19) into the primary composition operation $A_{\mathbb{M}}$ (8):

$$A_{\mathbb{M}}(f)(x) = \lambda c.\, f\big(\lambda g.\, x\big(\lambda y.\, c(g(y))\big)\big) : \mathbb{M}\beta \qquad \forall f : \mathbb{M}(\alpha \to \beta),\ x : \mathbb{M}\alpha. \tag{21}$$

Letting $f = \eta(\text{smoke})$ and $x = \eta(\text{John})$ then gives

$$\begin{aligned}[\![\textit{John smokes}]\!] &= \lambda c.\, \eta(\text{smoke})\big(\lambda g.\, \eta(\text{John})\big(\lambda y.\, c(g(y))\big)\big)\\ &= \lambda c.\, \eta(\text{John})\big(\lambda y.\, c(\text{smoke}(y))\big) = \lambda c.\, c(\text{smoke}(\text{John})) : \mathbb{M}t.\end{aligned}$$

In the second step above, note how the term $\lambda y.\, c(\text{smoke}(y))$ represents the future for the computation of $[\![\textit{John}]\!]$, namely to check whether he smokes, then pass the result to the context c containing the clause. If *John smokes* is the main clause, then the context c is simply the identity function id_ω. We define an evaluation operator $\varepsilon : \mathbb{M}\omega \to \omega$ by $\varepsilon(m) = m(\text{id}_\omega)$. Fixing $\omega = t$, we then have $\varepsilon\big([\![\textit{John smokes}]\!]\big) = \text{smoke}(\text{John})$, as desired.

Continuing to fix $\omega = t$, we can specify a meaning for *everyone*:

$$[\![\textit{everyone}]\!] = \lambda c.\, \forall x.\, c(x) : \mathbb{M}e. \tag{22}$$

This formula is not of the form $\lambda c.\, c(\dots)$. In other words, the meaning of *everyone* non-trivially manipulates the continuation, and so cannot be obtained from applying η to an ordinary value. Using the continuized composition rule (21), we now compute a denotation for *everyone smokes*:

$$\begin{aligned}[\![\textit{everyone smokes}]\!] &= \lambda c.\, \eta(\text{smoke})\big(\lambda g.\, [\![\textit{everyone}]\!]\big(\lambda y.\, c(g(y))\big)\big)\\ &= \lambda c.\, [\![\textit{everyone}]\!]\big(\lambda y.\, c(\text{smoke}(y))\big) = \lambda c.\, \forall x.\, c(\text{smoke}(x)) : \mathbb{M}t,\end{aligned}$$

giving $\varepsilon\big([\![\textit{everyone smokes}]\!]\big) = \forall x.\,\mathrm{smoke}(x) : t$, as desired.

In Barker's account, quantifier scoping corresponds to *evaluation order*. The composition operation $A_{\mathbb{M}}$, as we defined it in (8), evaluates f (into a) before x (into b). This gives the inverse scope reading of *someone loves everyone*. If we replace the right hand side of (8) with

$$x \star \big[\lambda b.\, f \star [\lambda a.\, \eta(a(b))]\big], \tag{23}$$

then we would get the opposite evaluation order and the linear scope reading of *someone loves everyone*. (Of the monads presented in this paper, the continuation monad is the only *non-commutative* one—that is, the only monad where evaluation order makes a difference.)

The main theoretical advantage of this analysis is that it is a compositional, in-situ analysis that does not invoke quantifier raising. Moreover, note that a grammar continuized is still a grammar—the continuized composition rule (21) is perfectly interpretable using the standard machinery of Montague grammar. In particular, we do not invoke any type ambiguity or flexibility as proposed by Partee and Rooth (1983) and Hendriks (1993); the interpretation mechanism performs no type-shifting at "run-time".

This desirable property also holds of the other monadic analyses I have presented. For instance, in a grammar with intensionality, meanings that use intensionality (for example $[\![\textit{student}]\!]$) are identical in type to meanings that do not (for example $[\![\textit{is}]\!]$). The interpretation mechanism does not dynamically shift the type of *is* to match that of *student*.

It is worth relating the present analysis to the computer science literature. Danvy and Filinski (1990) studied *composable continuations*, which they manipulated using two operators "shift" and "reset". The meaning of *everyone* specified in (22) can be expressed in terms of shift. To encode scope islands, Barker implicitly used reset. Filinski (1999) proved that, in a certain sense, composable continuations can simulate monads.

3 Combining monads

Having placed various semantic phenomena in a monadic framework, we now ask a natural question: Can we somehow combine monads in a modular fashion to characterize interaction between semantic phenomena, for example between intensionality and quantification?

Unfortunately, there exists no general construction for composing two arbitrary monads, say $(\mathbb{M}_1, \eta_1, \star_1)$ and $(\mathbb{M}_2, \eta_2, \star_2)$, into a new monad of the form $(\mathbb{M}_3 = \mathbb{M}_1 \circ \mathbb{M}_2, \eta_3, \star_3)$ (King and Wadler 1993; Jones and Duponcheel 1993). One might still hope to specialize and combine monads with additional structure, to generalize and combine monads as instances of a broader concept, or even to find that obstacles in combining monads are reflected in semantic constraints in natural language.

Researchers in denotational semantics of programming languages have made several proposals towards combining monadic functionality, none of which are completely satisfactory (Moggi 1990; Steele 1994; Liang, Hudak, and Jones 1995; Espinosa 1995; Filinski 1999). In this section, I will relate one prominent approach to natural language semantics.

3.1 Monad morphisms

One approach to combining monads, taken by Moggi (1990), Liang et al. (1995), and Filinski (1999), is to compose *monad morphisms* instead of monads themselves. A monad morphism (also known as a *monad transformer* or a *monad layering*) is a map from monads to monads; it takes an arbitrary monad and transforms it into a new monad, presumably defined in terms of the old monad and supporting a new layer of functionality. For instance, given any monad $(\mathbb{M}_1, \eta_1, \star_1)$ and fixing a type ρ, the *reader monad morphism* constructs a new monad $(\mathbb{M}_2, \eta_2, \star_2)$, defined by

$$\begin{aligned} \mathbb{M}_2\alpha &= \rho \to \mathbb{M}_1\alpha && \forall\alpha, && (24a)\\ \eta_2(a) &= \lambda w.\, \eta_1(a) : \mathbb{M}_2\alpha && \forall a : \alpha, && (24b)\\ m \star_2 k &= \lambda w.\, \bigl[m(w) \star_1 \lambda a.\, k(a)(w)\bigr] : \mathbb{M}_2\beta && \forall m : \mathbb{M}_2\alpha,\ k : \alpha \to \mathbb{M}_2\beta. && (24c) \end{aligned}$$

If we let the old monad $(\mathbb{M}_1, \eta_1, \star_1)$ be the *identity monad*, defined by $\mathbb{M}_1\alpha = \alpha$, $\eta_1(a) = a$, and $m \star_1 k = k(m)$, then the new monad (24) is just the reader monad (16). If we let the old monad be some other monad—even the reader monad itself—the new monad adds reader functionality.

By definition, each monad morphism must specify how to embed computations inside the old monad into the new monad. More precisely, each monad morphism must provide a function (pronounced "lift")

$$\ell : \mathbb{M}_1\alpha \to \mathbb{M}_2\alpha, \tag{25}$$

polymorphic in α. By definition, ℓ must satisfy naturality:

$$\begin{aligned} \ell\bigl(\eta_1(a)\bigr) &= \eta_2(a) && \forall a : \alpha, && (26a)\\ \ell\bigl(m \star_1 k\bigr) &= \ell(m) \star_2 (\ell \circ k) && \forall m : \mathbb{M}_1\alpha,\ k : \alpha \to \mathbb{M}_1\beta. && (26b) \end{aligned}$$

For the reader monad morphism, ℓ is defined by

$$\ell(m) = \lambda w.\, m : \mathbb{M}_2\alpha \qquad \forall m : \mathbb{M}_1\alpha. \tag{27}$$

The continuation monad also generalizes to a monad morphism. Fixing an answer type ω, the *continuation monad morphism* takes any monad $(\mathbb{M}_1, \eta_1, \star_1)$ to the monad $(\mathbb{M}_2, \eta_2, \star_2)$ defined by

$$\begin{aligned} \mathbb{M}_2\alpha &= (\alpha \to \mathbb{M}_1\omega) \to \mathbb{M}_1\omega && \forall\alpha, && (28a)\\ \eta_2(a) &= \lambda c.\, c(a) : \mathbb{M}_2\alpha && \forall a : \alpha, && (28b)\\ m \star_2 k &= \lambda c.\, m\bigl(\lambda a.\, k(a)(c)\bigr) : \mathbb{M}_2\beta && \forall m : \mathbb{M}_2\alpha,\ k : \alpha \to \mathbb{M}_2\beta. && (28c) \end{aligned}$$

The lifting function ℓ for the continuation monad morphism is defined by

$$\ell(m) = \lambda c.\, (m \star_1 c) : \mathbb{M}_2\alpha \qquad \forall m : \mathbb{M}_1\alpha. \tag{29}$$

Under the monad morphism approach, a monad can be transformed through a monad morphism to give a combined monad that supports the functionality of both the monad and the monad morphism. Since each monad morphism is a map from arbitrary monads to other monads, monad morphisms can be freely composed with each other, though the order of composition is significant. Applying the continuation monad morphism to the reader monad is equivalent to applying to the identity monad the composition of the continuation monad morphism and the reader monad morphism, and yields a monad with type constructor $\mathbb{M}\alpha = (\alpha \to \rho \to \omega) \to \rho \to \omega$. Applying the reader monad morphism to the continuation monad is equivalent to applying to the identity monad the composition of the reader monad morphism and the continuation monad morphism, and yields a different monad, with type constructor $\mathbb{M}\alpha = \rho \to (\alpha \to \omega) \to \omega$.

3.2 Translating monads to monad morphisms

The monad morphisms (24) and (28) may appear mysterious, but we can in fact obtain them from their monad counterparts (16) and (19) via a mechanical translation. The translation takes a monad $(\mathbb{M}_0, \eta_0, \star_0)$ whose η_0 and $\star_0$ operations are λ-terms, and produces a morphism mapping any old monad $(\mathbb{M}_1, \eta_1, \star_1)$ to a new monad $(\mathbb{M}_2, \eta_2, \star_2)$. Informally speaking, the translation treats the λ-calculus with which $(\mathbb{M}_0, \eta_0, \star_0)$ is defined as a programming language with side effects, and specifies a denotational semantics for it.

At the beginning of §3, we mentioned that there is no way to compose two arbitrary monads. This entails that there is no way to translate an arbitrary monad to a monad morphism. The translation we are about to describe is defined recursively on the structure of λ-types and λ-terms. It thus requires $\mathbb{M}_0$ to be defined as a λ-type, and η_0 and $\star_0$ to be defined as λ-terms. Therefore, the translation cannot apply to the powerset and pointed powerset monads (see footnote 1). Nevertheless, any monad morphism (including ones produced by the translation) can be applied to any monad (including these two monads).

Every type τ is either a function type or a base type. A function type has the form $\tau_1 \to \tau_2$, where τ_1 and τ_2 are types. A base type ι is a type fixed in $\mathbb{M}_0$ (ρ and ω in our cases), a polymorphic type variable (α and β as appearing in η and $\star$ (2) and ℓ (25)), or the terminal type ! (also known as the unit type or the void type). For every type τ, we recursively define its *computation translation* $\lceil \tau \rceil$ and its *value translation* $\lfloor \tau \rfloor$:

$$\lceil \iota \rceil = \mathbb{M}_1 \iota, \qquad \lfloor \iota \rfloor = \iota, \qquad \lceil \tau_1 \to \tau_2 \rceil = \lfloor \tau_1 \to \tau_2 \rfloor = \lfloor \tau_1 \rfloor \to \lceil \tau_2 \rceil, \tag{30}$$

where ι is any base type. Intuitively, each type τ in the λ-calculus with which $(\mathbb{M}_0, \eta_0, \star_0)$ is defined corresponds to two types in the pure λ-calculus: the type $\lceil \tau \rceil$ of τ-computations, and the type $\lfloor \tau \rfloor$ of τ-values.

Each term $e : \tau$ is an application term, an abstraction term, a variable term, or the terminal term. An application term has the form $(e_1 : \tau_1 \to \tau_2)\ (e_2 : \tau_1) : \tau_2$, where e_1 and e_2 are terms. An abstraction term has the form $(\lambda x : \tau_1 .\, e : \tau_2) : \tau_1 \to \tau_2$, where e is a term. A variable term has the form $x : \tau$, where x is the name of a variable of type τ. The terminal term is $! : !$ and represents the unique value of the terminal type $!$. For every term $e : \tau$, we recursively define its *term translation* $\lceil e \rceil : \lceil \tau \rceil$:

$$\begin{aligned}
&\big\lceil (e_1 : \iota \to \tau_1 \to \cdots \to \tau_n \to \iota')(e_2 : \iota) \big\rceil = \\
&\qquad \lambda y_1 : \lfloor \tau_1 \rfloor . \ldots \lambda y_n : \lfloor \tau_n \rfloor . \, \big[\lceil e_2 \rceil \star_1 \big(\lambda y_0 : \iota . \lceil e_1 \rceil (y_0) \ldots (y_n) \big) \big], && \text{(31a)}\\
&\big\lceil (e_1 : (\tau_1 \to \tau_2) \to \tau_3)(e_2 : \tau_1 \to \tau_2) \big\rceil = \lceil e_1 \rceil \big(\lceil e_2 \rceil \big), && \text{(31b)}\\
&\lceil \lambda x : \tau . \, e \rceil = \lambda x : \lfloor \tau \rfloor . \lceil e \rceil , && \text{(31c)}\\
&\lceil x : \iota \rceil = \eta_1(x), && \text{(31d)}\\
&\lceil x : \tau_1 \to \tau_2 \rceil = x, && \text{(31e)}\\
&\lceil ! \rceil = \eta_1(!), && \text{(31f)}
\end{aligned}$$

where ι and ι' are any base types, and $y_0, \ldots, y_n$ are fresh variable names.

Finally, to construct the new monad $(\mathbb{M}_2, \eta_2, \star_2)$, we specify

$$\begin{gathered}
\mathbb{M}_2 \alpha = \lceil \mathbb{M}_0 \alpha \rceil , \qquad \eta_2 = \lceil \eta_0 \rceil , \qquad \star_2 = \lceil \star_0 \rceil , \\
\ell(m) = \big\lceil \lambda f : ! \to \alpha . \, \eta_0(f(!)) \big\rceil \, (\lambda ! : ! . \, m) \quad \forall m : \mathbb{M}_1 \alpha .
\end{gathered} \tag{32}$$

To illustrate this translation, let us expand out $\mathbb{M}_2$ and $\star_2$ in the special case where $(\mathbb{M}_0, \eta_0, \star_0)$ is the reader monad (16). From the type translation rules (30) and the specification of $\mathbb{M}_2$ in (32), we have

$$\mathbb{M}_2 \alpha = \lceil \rho \to \alpha \rceil = \lfloor \rho \rfloor \to \lceil \alpha \rceil = \rho \to \mathbb{M}_1 \alpha ,$$

matching (24a) as desired. From the term translation rules (31) and the specification of $\star_2$ in (32), we have

$$\begin{aligned}
\star_2 &= \big\lceil \lambda m : \rho \to \alpha . \, \lambda k : \alpha \to \rho \to \beta . \, \lambda w : \rho . \, k\big(m(w)\big)(w) \big\rceil && \text{by (16c)}\\
&= \lambda m : \rho \to \mathbb{M}_1 \alpha . \, \lambda k : \alpha \to \rho \to \mathbb{M}_1 \beta . \, \lambda w : \rho . \, \big\lceil k\big(m(w)\big)(w) \big\rceil && \text{by (31c),}
\end{aligned}$$

in which

$$\begin{aligned}
\big\lceil k\big(m(w)\big)(w) \big\rceil &= \eta_1(w) \star_1 \lambda y_0 : \rho . \, \big\lceil k\big(m(w)\big) \big\rceil \, (y_0) && \text{by (31a), (31d)}\\
&= \big\lceil k\big(m(w)\big) \big\rceil \, (w) && \text{by (3a)}\\
&= \big(\lceil w \rceil \star_1 \lambda y_0 : \rho . \, \lceil m \rceil \, (y_0) \big) \star_1 \lambda y_0 : \alpha . \, \lceil k \rceil \, (y_0)(w) && \text{by (31a)}\\
&= \big(\eta_1(w) \star_1 \lambda y_0 : \rho . \, m(y_0) \big) \star_1 \lambda y_0 : \alpha . \, k(y_0)(w) && \text{by (31d), (31e)}\\
&= m(w) \star_1 \lambda y_0 : \alpha . \, k(y_0)(w) && \text{by (3a),}
\end{aligned}$$

matching (24c) as desired.

As mentioned earlier, the intuition behind our translation is to treat the λ-calculus with which $(\mathbb{M}_0, \eta_0, \star_0)$ is defined as a programming language whose terms may have computational side effects. Our translation specifies a semantics for this programming language in terms of $(\mathbb{M}_1, \eta_1, \star_1)$ that is call-by-value and that allows side effects only at base types. That the semantics is call-by-value rather than call-by-name is reflected in the type translation rules (30), where we define $\lfloor \tau_1 \to \tau_2 \rfloor$ to be $\lfloor \tau_1 \rfloor \to \lceil \tau_2 \rceil$ rather than $\lceil \tau_1 \rceil \to \lceil \tau_2 \rceil$. That side effects occur only at base types is also reflected in the rules, where we define $\lceil \tau_1 \to \tau_2 \rceil$ to be $\lfloor \tau_1 \to \tau_2 \rfloor$ rather than $\mathbb{M}_1 \lfloor \tau_1 \to \tau_2 \rfloor$. Overall, our translation is a hybrid between the call-by-name Algol translation (Benton, Hughes, and Moggi 2000; §3.1.2) and the standard call-by-value translation (Wadler 1992a; §8; Benton et al. 2000; §3.1.3).

3.3 A call-by-name translation of monads

Curiously, the semantic types generated by monad morphisms seem sometimes not powerful enough. As noted at the end of §3.1, the reader and continuation monad morphisms together give rise to two different monads, depending on the order in which we compose the monad morphisms. Fixing $\rho = s$ for the reader monad morphism and $\omega = t$ for the continuation monad morphism, the two combined monads have the type constructors

$$\mathbb{M}_{cr}\alpha = (\alpha \to s \to t) \to s \to t, \qquad \mathbb{M}_{rc}\alpha = s \to (\alpha \to t) \to t. \tag{33}$$

Consider now sentences such as

John wanted to date every professor (at the party). (34)

This sentence has a reading where every professor at the party is a person John wanted to date, but John may not be aware that they are professors. On this reading, note that the world where the property of professorship is evaluated is distinct from the world where the property of dating is evaluated. Therefore, assuming that *to date every professor* is a constituent, its semantic type should mention s in contravariant position at least twice. Unfortunately, the type constructors $\mathbb{M}_{cr}$ and $\mathbb{M}_{rc}$ (33) each have only one such occurrence, so neither $\mathbb{M}_{cr}t$ nor $\mathbb{M}_{rc}t$ can be the correct type.

Intuitively, the semantic type of *to date every professor* ought to be

$$\big((s \to t) \to s \to t\big) \to s \to t \tag{35}$$

or an even larger type. The type (35) is precisely equal to $\mathbb{M}_{cr}(s \to t)$, but simply assigning $\mathbb{M}_{cr}(s \to t)$ as the semantic type of *to date every professor* would be against our preference for the reader monad morphism to be the only component of the semantic system that knows about the type s. Instead, what we would like is to equip the transformation on types taking each α to $\big((s \to \alpha) \to s \to t\big) \to s \to t$ with a composition method for transformed values.

One tentative idea for synthesizing such a composition method is to replace the call-by-value translation described in §3.2 with a call-by-name translation, such as the Algol translation mentioned earlier (Benton et al. 2000; §3.1.2).[2] For every type τ, this translation recursively defines a type $[\![\tau]\!]$:

$$[\![\iota]\!] = \mathbb{M}_1\iota, \qquad [\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!], \tag{36}$$

where ι is any base type. For every term $e : \tau$, this translation recursively defines a term $[\![e]\!] : [\![\tau]\!]$:

$$\begin{aligned} [\![e_1(e_2)]\!] &= [\![e_1]\!]\bigl([\![e_2]\!]\bigr), & [\![x]\!] &= x, \\ [\![\lambda x : \tau.\, e]\!] &= \lambda x : [\![\tau]\!].\, [\![e]\!], & [\![!]\!] &= \eta_1(!). \end{aligned} \tag{37}$$

Applying this translation to a monad $(\mathbb{M}_0, \eta_0, \star_0)$ gives the types

$$[\![\eta_0]\!] : \mathbb{M}_1\alpha \to [\![\mathbb{M}_0\alpha]\!], \tag{38a}$$

$$[\![\star_0]\!] : [\![\mathbb{M}_0\alpha]\!] \to \bigl(\mathbb{M}_1\alpha \to [\![\mathbb{M}_0\beta]\!]\bigr) \to [\![\mathbb{M}_0\beta]\!]. \tag{38b}$$

If we let $(\mathbb{M}_0, \eta_0, \star_0)$ be the continuation monad (19) and $(\mathbb{M}_1, \eta_1, \star_1)$ the reader monad (16), then the type t transformed is

$$[\![\mathbb{M}_0 t]\!] = (\mathbb{M}_1 t \to \mathbb{M}_1 t) \to \mathbb{M}_1 t = \bigl((s \to t) \to s \to t\bigr) \to s \to t,$$

as desired. However, unless $(\mathbb{M}_1, \eta_1, \star_1)$ is the identity monad, the types in (38) do not match the definition of monads in (2). In other words, though our call-by-name translation does give the type transform we want as well as a composition method in some sense, its output is not a monad morphism.

4 Conclusion

In this paper, I used monads to characterize the similarity between several semantic accounts—for interrogatives, focus, intensionality, variable binding, and quantification.[3] In each case, the same monadic composition rules and mostly the same lexicon were specialized to a different monad. The monad primitives η and $\star$ recur in semantics with striking frequency.

It remains to be seen whether monads would provide the appropriate conceptual encapsulation for a semantic theory with broader coverage. In particular, for both natural and programming language semantics, combining monads—or perhaps monad-like objects—remains an open issue that promises additional insight.

[2] Another possible translation is the standard ("Haskell") call-by-name one (Wadler 1992a; §8; Benton et al. 2000; §3.1.1). It produces strictly larger types than the Algol call-by-name translation, for instance $s \to \bigl(s \to (s \to \alpha) \to s \to t\bigr) \to s \to t$.

[3] Other phenomena that may fall under the monadic umbrella include presuppositions (the error monad) and dynamic semantics (the state monad).

References

Barker, Chris. 2001. Continuations: In-situ quantification without storage or type-shifting. In *SALT XI: Semantics and linguistic theory*, ed. Rachel Hastings, Brendan Jackson, and Zsofia Zvolensky. Ithaca: Cornell University Press.

Benton, Nick, John Hughes, and Eugenio Moggi. 2000. Monads and effects. Lecture notes, International Summer School on Applied Semantics; 5 Sep. 2000, `http://www.disi.unige.it/person/MoggiE/APPSEM00/`.

Danvy, Olivier, and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

Espinosa, David A. 1995. Semantic lego. Ph.D. thesis, Graduate School of Arts and Sciences, Columbia University.

Filinski, Andrzej. 1999. Representing layered monads. In *POPL '99: Conference record of the annual ACM symposium on principles of programming languages*, 175–188. New York: ACM Press.

Hamblin, C. L. 1973. Questions in Montague English. *Foundations of Language* 10:41–53.

Heim, Irene, and Angelika Kratzer. 1998. *Semantics in generative grammar.* Oxford: Blackwell.

Hendriks, Herman. 1993. Studied flexibility: Categories and types in syntax and semantics. Ph.D. thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Jones, Mark P., and Luck Duponcheel. 1993. Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Department of Computer Science, Yale University, New Haven.

Kelsey, Richard, William Clinger, Jonathan Rees, et al. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation* 11(1):7–105. Also as *ACM SIGPLAN Notices* 33(9):26–76.

King, David J., and Philip Wadler. 1993. Combining monads. In *Functional programming, Glasgow 1992: Proceedings of the 1992 Glasgow workshop on functional programming*, ed. John Launchbury and Patrick M. Sansom. Berlin: Springer-Verlag.

Lappin, Shalom, ed. 1996. *The handbook of contemporary semantic theory.* Oxford: Blackwell.

Liang, Sheng, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *POPL '95: Conference record of the annual ACM symposium on principles of programming languages*, 333–343. New York: ACM Press.

Moggi, Eugenio. 1990. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

———. 1991. Notions of computation and monads. *Information and Computation* 93(1): 55–92.

Montague, Richard. 1974. The proper treatment of quantification in ordinary English. In *Formal philosophy: Selected papers of Richard Montague*, ed. Richmond Thomason, 247–270. New Haven: Yale University Press.

Partee, Barbara. 1996. The development of formal semantics. In Lappin (1996), 11–38.

Partee, Barbara, and Mats Rooth. 1983. Generalized conjunction and type ambiguity. In *Meaning, use and interpretation of language*, ed. Rainer Bäuerle, Christoph Schwarze, and Arnim von Stechow, 361–383. Berlin: de Gruyter.

Rooth, Mats. 1996. Focus. In Lappin (1996), 271–297.

Rooth, Mats Edward. 1985. Association with focus. Ph.D. thesis, Department of Linguistics, University of Massachusetts.

Steele, Guy L., Jr. 1994. Building interpreters by composing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 472–492. New York: ACM Press.

Wadler, Philip. 1992a. Comprehending monads. *Mathematical Structures in Computer Science* 2(4):461–493.

———. 1992b. The essence of functional programming. In *POPL '92: Conference record of the annual ACM symposium on principles of programming languages*, 1–14. New York: ACM Press.