Dynamic semantics as monadic computation

Christina Unger

CITEC, Bielefeld University

Abstract. This paper proposes a formulation of the basic ideas of dynamic semantics in terms of the state monad. Such a monadic treatment allows to specify meanings as computations that clearly separate context updating and context accessing operations from purely truth conditional meaning composition. Different behavior regarding the availability of referents throughout a discourse is modelled by adding structure to states, more specifically by distinguishing between global and local contexts, while relying solely on basic operations on sets and stacks.

1 Introduction

In the Montegovian tradition, formal semantics of natural languages are formulated in terms of the lambda calculus, starting with a core set of types, lexical meanings and simple composition rules. To account for phenomena such as intensionality, new types are introduced and make it necessary to revise all existing lexical meanings and composition rules in order to incorporate the new meaning aspect. In order to simplify presentations and allow for uniform, compositional and modular analyses of different phenomena, Shan [6] proposed to phrase formal semantic accounts in terms of monads.

The concept of monads stems from category theory and became a key tool for structuring the denotational semantics of programming languages [4] as well as for modelling computational effects such as non-determinism, continuations, state changes, exceptions and input-output [7]. Some of these concepts have also been applied to the semantics of natural language, e.g. continuations for a treatment of quantification [1] and exception handling for capturing presupposition projection [3].

Shan [6] considers several monads well-suited for capturing semantic phenomena: the (pointed) powerset monad for interrogatives and focus, the reader monad for intensionality, and the continuation monad for quantification. Furthermore there is a reasonable consensus that dynamic semantics can be phrased in terms of the state monad, representing common wisdom of dynamic semantic theories as stateful computations. Such a treatment was, e.g., provided by Ogata [5] and Bekki [2]. This paper proposes a slightly different way to do this, mainly relying on the structure of the state in order to capture different context updating and accessing behaviors.

A monadic approach has two benefits. The first one is a clear separation of those meaning aspects that affect the context from static meaning aspects in a way that retains full compositionality. The second one is modularity. Since all

monads rely on the same primitives and composition rules, our state monad for dynamic semantics can be composed with monads capturing other phenomena such as intensionality and presuppositions in a modular fashion.

2 The state monad

A monad is a triple $(M, \texttt{unit}, \star)$, where M is a type constructor mapping each type α to the corresponding monadic type $M\alpha$ (objects of type $M\alpha$ can be thought of as computations that yield a value of type α), `unit` is a function of type $\alpha \to M\alpha$ that injects the value into the monad (i.e. it transforms a value into a computation), and $\star$ (pronounced 'bind') is a function of type $M\alpha \to (\alpha \to M\beta) \to M\beta$ that composes two computations, where the second one depends on a value yielded by the first one.

The state monad represents computations that read and modify a state, where a state can be any kind of environment: a counter, a tree, a set of entities, and so on. The type constructor M in the case of the state monad constructs a function type that takes a state as input and returns a pair of a value and a (possibly new or modified state) as output:

$$M\alpha = \mathit{State} \to (\alpha \times \mathit{State})$$

We take *State* to be a type synonym for a set of entities, representing the context that stores anaphoric possibilities in dynamic semantics. We will therefore use variables $c, c', \ldots$ for states.

The functions `unit` and $\star$ of the state monad are defined as follows:

$$\begin{aligned} \texttt{unit}\ x &= \lambda c.\langle x, c\rangle \\ v \star k &= \lambda c.k\ \pi_1(v\ c)\ \pi_2(v\ c) \end{aligned}$$

Where π_1 and π_2 are functions that return the first and second element of a pair, respectively.

Monadic function application @ of type $M(\alpha \to \beta) \to M\alpha \to M\beta$ is commonly defined as follows:

$$k \mathbin{@} v = k \star \lambda f.(v \star \lambda x.\texttt{unit}\ (f\ x))$$

This can be read as the following sequence of computation steps: Compute k and name the result f, compute v and name the result x, then apply f to x and inject the result into the monad again. For the state monad, $k \mathbin{@} v$ reduces to $\lambda c.\langle f\ x, c\rangle$, i.e. the result of extracting the function and its argument from the monad, applying the former to the latter and injecting the result into the monad again.

For practical reasons, we additionally define a function $\triangleright$ of type $M\alpha \to M\beta \to M\beta$ for threading operations that only affect the state without producing a meaningful value, defined as follows: $k \triangleright v = k \star \lambda x.v$, where x must not occur free in v. Introducing entities into the context will be an example for such an operation.

3 Formulating dynamic semantics in terms of the state monad

We first inject the familiar denotations of nouns, verbs, etc. into the state monad, i.e. every denotation of type α will be lifted to a denotation of type $M\alpha = \mathit{State} \to \alpha \times \mathit{State}$. Then we will specify operations reading and updating the state and add them to the denotations of proper names and pronouns. Finally, we will hint at additions necessary for capturing quantifiers as well.

3.1 Lifting denotations to the state monad

Values are injected into a monad by means of the function `unit`. We thus start from the familiar denotations and lift them to monadic computations by applying `unit`. For the denotation of a common noun like unicorn of type $e \to t$ we thus get a monadic denotation of type $M(e \to t)$, i.e. $\mathit{State} \to (e \to t) \times \mathit{State}$:

$$[\![\mathsf{unicorn}]\!] = \texttt{unit}\ (\lambda x.\mathit{unicorn}\ x) = \lambda c.\langle \lambda x.\mathit{unicorn}\ x,\ c\rangle$$

Since we will use generalized quantifiers as noun phrase denotations, we use lifted verb denotations of type $gq \to gq \to t$, where gq is short for $(e \to t) \to t$. They are injected into the monad with `unit`, as before:

$$[\![\mathsf{whistles}]\!] = \texttt{unit}\ (\lambda \mathcal{P}.\mathcal{P}\ (\lambda x.\mathit{whistle}\ x))$$
$$[\![\mathsf{admires}]\!] = \texttt{unit}\ (\lambda \mathcal{P} \lambda \mathcal{Q}.\mathcal{P}\ (\lambda x.\mathcal{Q}\ (\lambda y.\mathit{admire}\ x\ y)))$$

Now suppose we lift the denotations of proper names such as Alice in the same way, such that $[\![\mathsf{Alice}]\!] = \texttt{unit}\ (\lambda P.P\ a)$. Then we can compute the meaning of Alice whistles by means of monadic function application:

$$\begin{aligned}
&[\![\mathsf{whistles}]\!]\ @\ [\![\mathsf{Alice}]\!] \\
=&\lambda c.\langle \lambda \mathcal{P}.\mathcal{P}\ (\lambda x.\mathit{whistle}\ x),\ c\rangle \star \lambda f.(\lambda c.\langle \lambda P.P\ a,\ c\rangle \star \lambda x.\texttt{unit}\ (f\ x)) \\
=&\lambda c.\langle \mathit{whistle}\ a,\ c\rangle
\end{aligned}$$

Thus, at the core of meaning computation nothing changes yet. We just introduced a context paramater and used `unit` and @ to hide this parameter and its threading. But what we actually want a proper name denotation to do is to introduce a new entity into the context that can be picked up by pronouns later on, for example in a discourse like Alice whistles. Bob admires her. We therefore need a way to modify the state.

3.2 State changing denotations

In order to add entities to and extract them from a context, we introduce two functions over contexts: $\hat{}$ of type $e \to \mathit{State} \to \mathit{State}$ that adds some entity x

to a context c with $c\hat{}x$ being the enriched context, and a function `sel` of type $\mathit{State} \to e$ that selects an entity from a context.[1]

Now, in order to read and modify the context, we define two state changing operations. The function `new` of type $e \to M\,()$ adds an entity to the context and returns the unit value $\_$ (actually it does not matter which value is returned, since we will thread this operation using $\triangleright$, which swallows the value):

$$\texttt{new}\ x = \lambda c.\langle \_, c\hat{}x\rangle$$

The function `get` of type $(e \to M(e \to t)) \to M(e \to t)$ replaces an argument position of type e by an entity selected from the context:

$$\texttt{get}\ m = \lambda c.m\ (\texttt{sel}\ c)\ c$$

Now we can specify the denotations of proper names and pronouns as follows:

$$[\![\mathsf{Alice}]\!] = (\texttt{new}\ a) \triangleright (\texttt{unit}\ \lambda P.P\ a) = \lambda c.\langle \lambda P.P\ a, c\hat{}a\rangle$$
$$[\![\mathsf{her}]\!] = \texttt{get} \triangleright (\lambda x.\texttt{unit}\ \lambda P.P\ x) = \lambda c.\langle \lambda P.P\ (\texttt{sel}\ c), c\rangle$$

That is, we inject the familiar denotations into the monad using `unit` and additionally compose it with a state affecting operation.

3.3 From sentences to discourses

For sequencing sentences we specify a merge operation $\oplus$ of type $Mt \to Mt \to Mt$ that composes two sentences, where the second one should be interpreted w.r.t. the context that the first one returns, and the returned value should be the conjunction of the two sentence meanings. Since sequencing is already encoded in $\star$, $\oplus$ can be defined straightforwardly (and in a way very similar to @):

$$s_1 \oplus s_2 = s_1 \star \lambda p.(s_2 \star \lambda q.(\texttt{unit}\ (p \wedge q)))$$

This definition can be read as follows: Compute s_1 and name the result p, compute s_2 and name the result q, then build the conjunction of p and q and inject it into the monad again.

Take, for example, the sentences Alice whistles and Bob admires her. The discourse of the former followed by the latter yields the following result:

$$\begin{aligned}&\mathsf{Alice\ whistles} \oplus \mathsf{Bob\ admires\ her}\\ =&\lambda c.\langle \mathit{whistle}\ a, c\hat{}a\rangle \oplus \lambda c.\langle \mathit{admire}\ (\texttt{sel}\ c)\ b, c\hat{}b\rangle\\ =&\lambda c.\langle (\mathit{whistle}\ a) \wedge (\mathit{admire}\ (\texttt{sel}\ c\hat{}a)\ b), c\hat{}a\hat{}b\rangle\end{aligned}$$

That is, Alice whistles is interpreted w.r.t. the input context c and updates it by adding a. The subsequent sentence Bob admires her is then interpreted w.r.t.

[1] Which entity is selected from the context should be determined by a pronoun resolution mechanism, which is out of the scope of this paper. We therefore assume `sel` to function as an oracle here.

this updated context $c\hat{}a$ and adds another entity, b, which could be picked up by pronouns still to come.

That is, meaning composition proceeds as usual, and additionally an input context (c) is related with an output context ($c\hat{}a\hat{}b$). In terms of DRT, a simple DRS $[x_1, \ldots, x_n \mid C_1, \ldots, C_m]$ with discourse referents $x_1, \ldots, x_n$ and conditions $C_1, \ldots, C_m$ would correspond to the lambda term $\lambda c.\langle C_1 \wedge \ldots \wedge C_m,\ c\hat{}x_n\hat{}\ldots\hat{}x_1\rangle$, i.e. one like the resulting lambda term from above.

3.4 Adding structure to states

The most interesting problem still remains: Quantifiers like most and every introduce entities into the context that are not accessible beyond the scope of the quantifier. E.g. in Every unicorn is eating Bob's flowers. He adores it, the pronoun it cannot pick up the entity introduced by every unicorn. This suggests that we need a way to empty the context. However, since entities introduced by proper names such as Bob are usually accessible throughout whole discourses (he in the second sentence can pick up Bob as referent without a problem), not all of the context can be deleted.

The solution we want to pursue here is to add more structure to the state. States cannot simply be sets of entities anymore, rather they have to distinguish local from global contexts. Quantifiers could then add entities into the local context, which is emptied once the interpretation process leaves the scope of the quantifier, whereas proper names introduce entities into the global context, which is kept throughout the whole discourse. Now, since different quantifiers can have different scopes, we actually need a local context for every quantifier occurrence, so we will assume states to be a pair of a global context and a LIFO stack (denoted as $[\cdot]$) of local contexts, where contexts are sets of entities as before (denoted as $\{e\}$).

$$State = [\{e\}] \times \{e\}$$

Proper names introduce entities into the global context, quantifiers push a new local context on the stack and introduce an entity there. The two definitions of according functions are adapted versions of the function $\texttt{new} :: e \to M()$ that we used earlier:

$$\texttt{new}_{global}\ x = \langle \_, (\texttt{snd}\ c)\hat{}x\rangle$$
$$\texttt{new}_{local}\ x = \langle \_, \texttt{add}\ x\ (\texttt{top}\ (\texttt{push}\ (\texttt{fst}\ c)))\rangle$$

Where `fst` and `snd` are the usual functions for accessing the first and second element of a pair, $\texttt{top} :: [\alpha] \to \alpha$ is a function that retrieves the top-most element from a stack, $\texttt{push} :: [\alpha] \to [\alpha]$ pushes an empty context on a stack, and $\texttt{add} :: \alpha \to \{\alpha\} \to \{\alpha\}$ adds an element to a set.

Proper names use $\texttt{new}_{global}$, i.e. the denotation of Alice, for example, is the following:

$$[\![\text{Alice}]\!] = (\texttt{new}_{global}\ a) \triangleright (\texttt{unit}\ \lambda P.P\ a)$$

The interpretation of pronouns does not change at all:

$$[\![\mathsf{she}]\!] = \mathtt{get} \triangleright (\lambda x.\mathtt{unit}\ \lambda P.P\ x)$$

We assume that the function `sel` still acts as an oracle that, given a state, selects an entity now from the union of all contexts within this state.

Quantifiers like every, most, and so on, will use $\mathtt{new}_{local}$, i.e. add an empty local context on the stack and add the variable they introduce to that local context.

3.5 Quantifier denotations

Quantifier denotations differ from the denotations of proper names in that they introduce an entity only locally, and empty that local context once their scope is closed, so the introduced entity is not available as antecedent outside of the quantifier's scope. In order to capture this behavior, we assemble quantifier denotations using the following ingredients:

- the function $\mathtt{new}_{local}$ that introduces an entity into a new local context
- the usual quantifier denotation lifted into the monad, e.g. $\mathtt{unit}\ \lambda P\lambda Q.\forall x.P\ x \rightarrow Q\ x$
- a function `clear` that removes the local context of the quantifier from the stack

We assume that the function `clear` has a way to recognize the context that belongs to the quantifiers (e.g by adding an identifier to the stack elements, but the exact mechanism does not matter here). Additional to that, we do not assume anything else but the common stack operation `pop`, a function that removes the top-most element from a stack. This means that if the local context of the quantifier is the top-most one, it is simply removed, however if it is lower on the stack, all contexts above it have to be removed as well, in order to reach it. And this is exactly what we assume `clear` does.[2] Here we do not give a precise formalisation of all low-level operations involved in this kind of popping until a certain context is reached, but hope that the concept is clear enough. Now, assuming we have such function $\mathtt{pop}_{local} :: \mathit{State} \rightarrow \mathit{State}$ that accesses the stack of local contexts and pops all elements from this stack until (and including)

[2] In many cases, different quantifier scopes can be closed in the order they were introduced, which would make the clearing mechanism even simpler. Cases of quantifiers outscoping each other, however, would require a slightly more sophisticated mechanism of accessing and removing only a certain (possibly non-top-most) element from the stack. Another solution is to exploit the modularity that monads offer by importing a separate monadic treatment of quantifier scoping. Such a monadic treatment could be built using the continuation monad, e.g. modelling the continuation approach to quantification proposed by Barker [1]. We will leave this issue for another time.

the context that was introduced by the quantifier. Then we define the function $\texttt{clear} :: t \to Mt$ simply as a function that applies $\texttt{pop}_{local}$ to the state and leaves the value of the computation untouched:

$$\texttt{clear} = \lambda v \lambda c.\langle v, \texttt{pop}_{local}\ c\rangle$$

Putting together all ingredients for quantifier denotations, we would want something like the following, i.e. a computation that inserts the usual quantifier denotation (here for every) into the state monad and composes it with two stateful computations, one introducing the relevant variable into a new local context and one removing this context again:

$$(\texttt{new}_{local}\ x) \triangleright (\texttt{unit}\ \lambda P \lambda Q.\forall x.P\ x \to Q\ x) \star \texttt{clear}$$

However, specifying the denotation like this would have the effect of adding a local context and immediately removing it again. Instead, we want clear to apply only after the quantifier denotation received its arguments. In order to achieve this, we have to lift the denotation and specify it as follows:

$$\lambda \overline{P} \lambda \overline{Q}.((((\texttt{new}_{local}\ x) \triangleright \texttt{unit}\ \lambda P \lambda Q.\forall x.P\ x \to Q\ x) @ \overline{P}) @ \overline{Q}) \star \texttt{clear}$$

This denotation is no longer of type $M((e \to t) \to (e \to t) \to t)$ but of type $M(e \to t) \to M(e \to t) \to Mt$. It takes two arguments $\overline{P}$ and $\overline{Q}$ of type $M(e \to t)$ and feeds them to the quantifier denotation $(\texttt{new}_{local}\ x) \triangleright (\texttt{unit}\ \lambda P \lambda Q.\forall x.P\ x \to Q\ x)$ using monadic application @, and applies clear to the result. I.e. the monadic composition of the quantifier and its arguments is introduced explicitely into the quantifier denotation, thuse quantifiers can be seen as taking control over the involved computations. Nevertheless, nothing is added to the idea of sequencing the three involved ingredients.

Let us look at a simple example: Every contestant thinks he wins. The denotation of every was given above, the denotations of the other lexical items are as expected:

$$\begin{aligned}
[\![\text{contestant}]\!] &= \texttt{unit}\ \lambda x.\mathit{contestant}\ x \\
[\![\text{thinks}]\!] &= \texttt{unit}\ \lambda p \lambda y.\mathit{think}\ p\ y \\
[\![\text{he}]\!] &= \texttt{get} \triangleright \lambda z.\texttt{unit}\ \lambda P.P\ z \\
[\![\text{wins}]\!] &= \texttt{unit}\ \lambda x.\mathit{win}\ x
\end{aligned}$$

First we compute the meaning of the embedded verb phrase thinks he wins by means of the monadic application $[\![\text{thinks}]\!]@([\![\text{he}]\!]@[\![\text{wins}]\!])$. The result is the following:

1. $\lambda c.\langle \lambda y.\mathit{thinks}\ (\mathit{win}\ (\texttt{sel}\ c))\ y, c\rangle$

Next, applying the denotation of every to the denotation of contestant, we get:

2. $\lambda\overline{Q}.((\lambda c.\langle\lambda Q.\forall x.\mathit{contestant}\ x \to Q\ x, c\hat{}\{x\}\rangle)@\overline{Q}) \star \texttt{clear})$

Where $c\hat{}\{x\}$ stands for the state c with a local context $\{x\}$ added. Finally, we apply 2 to 1, which yields:

3. $\lambda c.\langle\forall x.\mathit{contestant}\ x \to \mathit{think}\ (\mathit{win}\ (\texttt{sel}\ c\hat{}\{x\}))\ x, c\rangle$

Note that the function `clear` removes the local context $\{x\}$ from the state, thus the state that will play a role in further computations is the input state c, while the state that the selection function `sel` applies to is still $c\hat{}\{x\}$.

At this point, we will not look into modelling varying scopal behaviors of different quantifiers, which would require a paper on its own (or two), but rather turn to indefinites, which exhibit a behavior that differs from quantifiers and requires another way of updating contexts.

3.6 Indefinites

In contrast to quantifiers such as every, most, no, and the like, indefinites like a unicorn differ in their behavior regarding the availability of introduced referents. In a simple predication without scope-taking elements such as quantifiers and negation, they are able to extend their scope arbitrarily far to the right, even across sentences as in 4.

4. Alice saw a unicorn in her garden. It was eating the flowers.

Because of this property, such free indefinites are often assumed to not be quantifiers. We follow this assumption and give existentials a meaning $\texttt{unit}\ \lambda P.P\ x$ with a free variable x that is supposed to be interpreted existentially once truth-conditions are assigned to a discourse. The scope of indefinites is, however, restricted if they occur in the scope of a quantifier, see 5. But inside this scope, they are free, as can be seen in a typical donkey-type sentence as 6.

5. Every formal semanticist saw a unicorn in his garden. #It was eating the flowers.

6. Every formal semanticist who saw a unicorn admired it.

That is, we want to capture that if there is a quantifier, the indefinite introduces its referent into that quantifier's local context, and if there is none, its referent is added to the global context where it is available throughout the discourse. To this end, we need a function slightly different from $\hat{}$ that decides to which context to add the introduced referent. We refert to it as + and propose

its working as follows: If there is a local context stack in state c, add a new variable x to the top-most[3] context of that stack: $\texttt{add}\ x\ (\texttt{top}\ (\texttt{fst}\ c))$. Otherwise add it to the global context: $\texttt{add}\ x\ (\texttt{snd}\ c)$. A rationale for this is that since existentials do not denote quantifiers, they do not have the force to open up a new local context, thus have to add their referent to an already existing context (be it local or global).

The denotation of indefinites now should contain the base denotation lifted into the monad: $\texttt{unit}\ \lambda P\lambda Q.P\ x \wedge Q\ x$, as well as the stateful computation that we call $\texttt{new}_{free}$:

$$(\texttt{new}_{free}\ x) \triangleright \texttt{unit}\ \lambda P\lambda Q.P\ x \wedge Q\ x$$

Where $\texttt{new}_{free}\ x = \lambda c.\langle \_, c + x\rangle$.

The goal is that the discourse in 7 receives the interpretation given in 8, where $c\hat{}x\hat{}a$ refers to the state c where x and a are added to the global context, and the discourse in 9 receives the interpretation given in 10, where $c\hat{}\{x, y\}$ refers to the state c with a local context containing x and y.

7. A unicorn barks at Alice. It is afraid.
8. $\lambda c.\langle \mathit{unicorn}\ x \wedge \mathit{barkAt}\ a\ x \wedge \mathit{afraid}\ (\texttt{sel}\ c\hat{}x\hat{}a), c\hat{}x\hat{}a\rangle$
9. Every gardener saw a unicorn.
10. $\lambda c.\langle \forall x.\mathit{gardener}\ x \rightarrow \mathit{unicorn}\ y \wedge \mathit{saw}\ y\ x, c\hat{}\{x, y\}\rangle \star \texttt{clear}$
    $= \lambda c.\langle \forall x.\mathit{gardener}\ x \rightarrow \mathit{unicorn}\ y \wedge \mathit{saw}\ y\ x, c\rangle$

The difference is that in 7, the stack of local contexts is empty, as it is the beginning of the discourse and no quantifier meaning was computed,[4] while in 9 the universal quantifier pushes a local context on the stack, to which the referent introduced by the existential will be added. Once all arguments of the universal quantifier are provided, the state is cleared, thereby also deleting y, which is thus not available as antecedent of later occurring pronouns.

Let us first look at example 7: A unicorn barks at Alice. It is afraid. In addition to the above denotation of the existential a, we have the following denotations:

$$\begin{aligned}
[\![\text{unicorn}]\!] &= \texttt{unit}\ \lambda x.\mathit{unicorn}\ x \\
[\![\text{barks at}]\!] &= \texttt{unit}\ \lambda x\lambda y.\mathit{barksAt}\ x\ y \\
[\![\text{Alice}]\!] &= (\texttt{new}_{global}\ a) \triangleright \texttt{unit}\ \lambda P.P\ a \\
[\![\text{it}]\!] &= \texttt{get} \triangleright \lambda x.\texttt{unit}\ \lambda P.P\ x \\
[\![\text{is afraid}]\!] &= \texttt{unit}\ \lambda x.\mathit{afraid}\ x
\end{aligned}$$

[3] The top-most stack is the one that is easiest accessible with the simple operations we assume, and empirically it seems warranted that existentials are available as antecedents only within the scope of the closest quantifier. E.g. the pronoun it in Most women like all men who saw a unicorn and admired it can refer to a unicorn only if the conjunction is interpreted as being within the scope of the universal all.

[4] Even if there was a quantifier in the earlier discourse, its local context would have been removed from the stack before encountering the first sentence of 7.

Composing the meaning of a unicorn barks at Alice yields the following:

$$\begin{aligned}&(\llbracket \mathsf{a} \rrbracket @ \llbracket \mathsf{unicorn} \rrbracket) @ (\llbracket \mathsf{barks\ at} \rrbracket @ \llbracket \mathsf{Alice} \rrbracket)\\ =&\lambda c.\langle \mathit{unicorn}\ x \wedge \mathit{barkAt}\ a\ x, (c + x)\hat{}a \rangle\end{aligned}$$

Assuming that there was no previous discourse, the input state c will contain no local context and an empty global context, so $(c + x)\hat{}a$ refers to the state where x and a are added to the global context. The denotation of It is afraid is the following:

$$\lambda c.\langle \mathit{afraid}\ (\mathtt{sel}\ c), c \rangle$$

Combining this with the denotation of the first sentence, we get:

$$\lambda c.\langle \mathit{unicorn}\ x \wedge \mathit{barkAt}\ a\ x \wedge \mathit{afraid}\ (\mathtt{sel}\ (c + x)\hat{}a), (c + x)\hat{}a \rangle$$

Let us now look at example 9 with an indefinite in the scope of a quantifier: Every gardener saw a unicorn. The denotation of the existential noun phrase a unicorn is $\lambda c.\langle \lambda Q.\mathit{unicorn}\ z \wedge Q\ z, c + z \rangle$, and combining it with the denotation of saw lifted in the second argument ($\mathtt{unit}\ \lambda \mathcal{P} \lambda y.\mathcal{P}\ \lambda x.\mathit{see}\ x\ y$) yields 11.

11. $\lambda c.\langle \lambda y.\mathit{unicorn}\ z \wedge \mathit{saw}\ z\ y, c + z \rangle$

Now applying the denotation of every first to the denotation of gardener (which is $\mathtt{unit}\ \lambda x.\mathit{gardener}\ x$) and then to 11 gives the sentence denotationa already provided in 10:

$$\begin{aligned}&\lambda c.\langle \forall x.\mathit{gardener}\ x \rightarrow \mathit{unicorn}\ y \wedge \mathit{saw}\ y\ x, c\hat{}\{x, y\} \rangle \star \mathtt{clear}\\ &\quad = \lambda c.\langle \forall x.\mathit{gardener}\ x \rightarrow \mathit{unicorn}\ y \wedge \mathit{saw}\ y\ x, c \rangle\end{aligned}$$

Treating indefinites in such a way, we compute their non-quantificational reading (there is some unicorn that every gardener saw). In order to also get the quantificational reading (for every gardener there is a possibly different unicorn), we would need a quantifier denotation of the existential, such as the following, completely in parallel to the denotation of every:

$$\lambda \overline{P} \lambda \overline{Q}.((((\mathtt{new}_{local}\ x) \triangleright \mathtt{unit}\ \lambda P \lambda Q.\exists x.P\ x \wedge Q\ x) @ \overline{P}) @ \overline{Q}) \star \mathtt{clear}$$

Finally, a note on negation: Negation is a scope-taking element which restricts the availability of referents introduced by indefinites just like quantifiers, as can be seen in 12.

12. Alice did not see a unicorn in her garden. #It was eating the flowers.

If we want to treat these cases analogously to cases like 9 above, negation has to push a local context on the state exactly like quantifiers, just without introducing a referent. This is achieved, e.g., by assuming the following denotation for sentence negation:

$$\llbracket \mathsf{not} \rrbracket = \lambda c.\langle \_, \mathtt{push}\ (\mathtt{snd}\ c) \rangle \triangleright \mathtt{unit}\ \lambda p.\neg p$$

Note that, as in the other cases, this denotation consists of the usual denotation lifted into the monad and composed with a purely state affecting computation, thus it separates the truth-conditional part of the meaning from the context changing one.

4 Conclusion

We used the state monad to formulate characteristics of dynamic semantics, where the state was assumed to consist of a global context and a stack of local contexts, with contexts being sets of referents. Different behavior of denotations that introduce referents was captured by whether they were introduced into the global or a local context and whether this context was cleared after computation or not. Proper names were assumed to add a referent to the global context, which is kept throughout the whole discourse, thus being available as antecedents without restriction. Quantifiers, on the other hand, were assumed to push a new local context with a newly added referent to the stack, and to clear this context once their computation is finished, thereby making their referent available as antecedent only within the quantifier's scope. Existentials constitute a case in the middle, being available arbitrarily long in a discourse if not under the scope of a quantifier or negation, otherwise being restricted to the closest embedding scope. We proposed to capture this behavior by assuming that free indefinites cannot introduce a new local context (due to not having quantificational force), thereby having to add a referent to an already existing context. We proposed that this context is the top-most local context if there is one, and the global context otherwise.

This monadic approach offers a computational view that separates stateful, i.e. context updating or context accessing operations (restricted to basic operations on sets and stacks), from static, truth conditional meaning composition.

Furthermore, a monadic treatment fits nicely into Shan's picture of a modular treatment of semantic phenoma such as intensionality, variable binding, presuppositions, and so on. How our proposed state monad would combine and interact with other monads, however, still remains to be worked out.

References

1. C. Barker. Continuations and the nature of quantification. *Natural Language Semantics*, 10:211–242, 2002.
2. D. Bekki. Monads and meta-lambda calculus. In H. Hattori et al., editor, *New Frontiers in Artificial Intelligence (JSAI 2008 Conference and Workshops, Asahikawa, Japan, June 2008, Revised Selected Papers from LENLS5) LNAI 5447*, pages 193–208. Springer, 2009.
3. P. de Groote and E. Lebedeva. Presupposition accommodation as exception handling. In *Proceedings of the 11th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, SIGDIAL '10, pages 71–74, 2010.
4. E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science, Asilomar, California*. IEEE, June 1989.

5. N. Ogata. Towards computational non-associative lambek lambda-calculi for formal pragmatics. In *Proceedings of the Fifth International Workshop on Logic and Engineering of Natural Language Semantics (LENLS2008) in Conjunction with the 22nd Annual Conference of the Japanese Society for Artificial Intelligence 2008*, pages 79–102, 2008.
6. C.-C. Shan. Monads for natural language semantics. In K. Striegnitz, editor, *Proceedings of the 2001 European Summer School in Logic, Language and Information Student Session*, pages 285–298, 2002.
7. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.