Monad Transformers for Natural Language: Combining Monads to Model Effect Interaction

Reuben Cohn-Gordon

Abstract

Following the work of (Shan 2002) and (Charlow 2014), we attempt to model the interactions of a number of natural language phenomena using the category theoretic apparatus of monads, and a device borrowed from functional programming allowing them to be stacked, the monad transformer. We identify six monads to stack, each corresponding to a phenomenon in NL semantics and establish the correct order in which to do so, to produce the appropriate interactions between them. We implement this grammar in Haskell, and consider a number of empirical predictions the system makes.

"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."   (Shan (2002))

Expanding on Montague's use of the simply typed lambda calculus for NL semantics, a series of papers by Shan and Barker show how modern techniques originating in the study of functional programming can be applied to natural language.

The use of monads provides an elegant and modular way to separate the core semantics from "effects", which in the domain of natural language amount to phenomena involving richer types than those provided by the simply typed lambda calculus alone. Each such type enrichment is then shown to correspond to a particular monad, often one which is familiar in the domain of functional programming. This approach is summed up by (Charlow 2014):

> Monadic Credo (after Lewis 1970): to say what meaning is, we may first ask what meanings can do, and then find a monad that lets them do that.

Following work such as (Giorgolo and Asudeh 2012), (Charlow 2014) and (Shan 2002), six such "effects" emerge which are particularly amenable to a monadic treatment[1]:

(1)
- Quantifier Scope: the Continuation Monad
- Conventional Implicature: the Writer Monad
- Presupposition Failure: the Exception Monad
- Anaphora: the State (and Set) Monad
- Intensionality: the Reader Monad
- Focus: the Pointed Set Monad

The monadic credo works well for individual type enrichments. However, it is imperative that our compositional grammar can handle multiple side effects at once, and successfully model the interactions between them. (Shan 2002) discusses the use of monad morphisms (known in the functional programming literature as monad transformers) to combine monadic effects.

Following this line of work, (Giorgolo and Asudeh 2012) makes use of monad transformers to explain apparent exceptions to the characterization of conventional implicature offered by (Potts 2005). In particular, sentences such as (2), defy the generalization that information should not flow from side-issue to at-issue meaning.

---

[1] Others exist, such as the Set monad for interrogatives, and even the IO monad for performatives, but we omit these from the current study.

(2)   Homer, who likes a child, teases him.

The insight of (Giorgolo and Asudeh 2012) is to show that this and similar effects can be modeled with monad transformers. This approach predicts that information inside a conventional implicature can "escape", so long as that information involves a monadic effect.

This simple case provides insight into the potential of monad transformers to give successful accounts of the subtle interactions between different natural language phenomena. Building on the work of (Charlow 2014) and others, we attempt to implement a grammar in Haskell, a common functional programming language, which models all six of the phenomena in (1).

This serves two purposes. Firstly, it puts the monad transformer based approach to NL semantics to the test on a larger scale than has previously been attempted. Secondly, it allows us to capture many empirical data of the ilk of (2), where an interaction between two monadic effects accounts for a subtle behavior.

The paper is structured as follows. We first introduce the fundamentals of monads and the mechanism for combining them. We then enter into a consideration of empirical phenomena, focusing for the time being on English data. As it turns out, there is a close relation between the technical question of how these effects should be combined, and the semantic question of how we want the phenomena to interact. Finally, we incrementally build a grammar with an increasing set of effects (the full version of which is available at https://github.com/reubenharry/reubenharry.github.io/tree/master/grammar).

1   Introducing Monads

Monads are in origin a mathematical construct, from the field of category theory. Because they are both very abstract and useful in functional programming, a common mode of introduction is the examination of disparate problems, the solution to each of which shares an abstract similarity. This similarity is what the notion of a monad is shown to capture. It therefore seems apt to introduce monads by first introducing the phenomena in natural language that they can model.

First, a few terminological notes. For clarity, and alignment of our definitions in this paper with the Haskell code that accompanies it, we denote function application "$f(x)$" as "fx". Likewise, for a higher order function f (e.g. a function of type (e → (e → t))), we will write (f x y) to denote its application to two arguments, which can be parsed right-associatively, as ((f x) y). Furthermore, (x,y) represents the ordered pair consisting of x as a first value and y as a second.

We use α and β to refer to arbitrary types in the simply typed lambda calculus. M is a *type constructor* if it takes any type and returns a new type. For instance, we can think of {} as a type constructor which takes a type α and returns a new type $\{\alpha\}$, i.e. a type contains a set of values of type α.

Any type constructor M that has functions *return* and *bind* of the following types is a monad:

return :: $\alpha \rightarrow$ M α

bind :: M $\alpha \rightarrow (\alpha \rightarrow$ M $\beta) \rightarrow$ M β

To see what this means in practice, let us turn to a brief discussion of each of the phenomena listed in (1):

1.1   Our Model

For the rest of the paper, it will prove useful to have a toy model from we can evaluate example sentences. We define it as follows, giving the characteristic sets for predicates[2].

- Domain : {Bart, Lisa, Homer, Maggie, Marge}

---

[2]A technical note: the characteristic sets, for predicates of more than one argument, are uncurried. That is to say, they are of type $(\alpha, \alpha...) \rightarrow$ t, rather than $(\alpha \rightarrow \alpha... \rightarrow$ t). This is just to make the definitions more straightforward.

- *child* : {Bart, Lisa, Maggie}

- *parent* : {Homer, Marge}

- *ran* : {Homer, Marge, Lisa, Bart}

- *loved* : {(Lisa, Homer), (Bart, Lisa), (Maggie, Homer), (Marge, Homer), (Homer, Homer), (Homer, Bart), (Marge, Lisa), (Homer, Maggie)}

- *stopped loving* : {(Lisa, Homer), (Maggie, Homer)}

For example, it is true that Lisa loved Bart, because (Bart, Lisa) is in the characteristic set of *loved*. Note that the order of the tuple appears backwards because verbs first take an object then a subject, compositionally.

1.2 Focus: Hamblin Semantics

A typical treatment of focus, marked prosodically in English, is to denote a focused element of a sentence by a set of alternative values. For example, the alternatives to $Bart_f$, in "$Bart_f$ is a Simpson." might be {Lisa, Homer}.

In conjunction with our model of section (1.1), sentence (3) has alternatives {False,False} and is itself True.

(3)   Lisa loved $Bart_f$

To model focus alternatives, we can introduce a new kind of type constructor in our system, the list[3]. For every type α, we declare (List α) to be a new type, which is the type of a list of values of type α. E.g. [Homer, Bart] would be of type (List e).

We say that the actual value is the leftmost one in the list, and the rest are the alternatives. For example, $Bart_f$ now denotes [Bart, Lisa, Homer], and (3) denotes [True,False,False].

Introducing List into our system leads to a proliferation of new types, (List e), List(e→t), e→(List t), and so on. What we now need is a way of doing composition with these new types, to explain how we can actually obtain values for (3), and other sentences. For example, suppose that ran_f is of type (e→(List t)) defined as[4]:

(4)   λx → {(ran x), not (ran x)}

Then the denotation of "$Bart_f$ ran_f." (a sentence with double focus, which is empirically attested) should be, with our model from section (1.1), [True,False,True,False,True,False]. The actual value, i.e. the value of "Bart ran" is the leftmost element of the list, True. How do we obtain this compositionally?

As it turns out, all we need to define are two functions, called *bind* and *return* which will let us do any sort of composition with our new types. This roughly amounts to Hamblin semantics. Read "::" as "is of type":

- bind :: (List α) → (α → (List β)) → (List β)

- bind x f = flatten ([(f y) for y in x])

This notation means the following: for each element in x, apply f to it, and list the results in order. This should give a list of lists. Then flatten this into a single list (an example of flatten: flatten([[1,2],[3,4]]) = [1,2,3,4])).

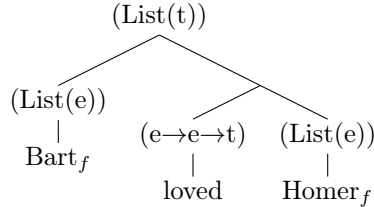- return :: α → (List α).

---

[3]We adapt this example from (Shan 2002), but use the list monad rather than the pointed set monad. This is strictly stronger, since lists preserve order among the alternatives, but this difference has little consequence here.

[4]The attentive reader might have noticed that we choose (e→List t) not (List (e→t)) for the type of ran_f. Here, the choice shouldn't make a different, but composition of (List e) with (e→List t) is more general that composition of (List e) with (List e→t), in a sense that will hopefully become clear. For more information, see the difference between applicatives and monads, discussed in section (3.1).

- return x = [x]

With these functions defined, we can compose our new types with relative ease. For example,"Bart_f ran_f" can be composed monadically in the expression (bind Bart_f ran_f).

More complex cases are also treatable in this way. For instance, consider the following doubly focused sentence:

$$
\begin{array}{c}
(\text{List(t)}) \\
\diagdown \\
(\text{List(e)}) \quad\quad (\text{e}\rightarrow\text{e}\rightarrow\text{t}) \quad (\text{List(e)}) \\
| \quad\quad\quad\quad | \quad\quad\quad\quad | \\
\text{Bart}_f \quad\quad \text{loved} \quad\quad \text{Homer}_f
\end{array}
$$

In this case, we could either perform monadic composition using the expression (5) or (6). For the List type constructor, there is no difference in the outcome between these two options, but this is not always the case, as we shall see.

(5)   bind Bart (λy $\rightarrow$ bind Homer (λx $\rightarrow$ return (loved x y)))

(6)   bind Homer (λx $\rightarrow$ bind Bart (λy $\rightarrow$ return (loved x y)))

1.2.1   Do-notation

Because of their ubiquity in Haskell, a convenient notation for using monads exists, known as *do notation*. This translates an expression like ((bind x (λy $\rightarrow$ return (f y)))) to

```
do
  y <- x
  return (f x)
```

This is particularly useful for cases like (5) and (6). Respectively, we can define denotations for each, labeled *bartFlovedHomer1* and *bartFlovedHomer2* as follows:

```
bartFlovedHomer1 = do
  y <- BartF
  x <- HomerF
  return (loved x y)
```

And:

```
bartFlovedHomer2 = do
  x <- HomerF
  y <- BartF
  return (loved x y)
```

We can understand the lines of the form "x $\leftarrow$ y" as a normal value of type α being "drawn out" of the monadic type (M α), and then composed on a successive line of the do-expression in such a way that the final line returns a monadic value of (M β). We will use this notation throughout the paper.

1.3 Presupposition Failure: the Exception Monad

Presupposition failure is another phenomenon which works its way up a compositional tree. For example, (7) triggers a presupposition failure for the whole sentence, as a result of the failure of "present king of Springfield", stemming from the fact that this entity does not exist:

(7) Bart teases the present king of Springfield.

We want a way to add types capable of representing sentences which are neither true nor false, but rather fail presuppositionally. We do so with another type constructor.

Our previous type constructor was List. We now consider Except[5]. The type (Except α) contains objects (i.e. values) such as (Left "[String]") or (Right α)[6]. This represents the fact that a value of type (Except α) can either be just a value of type α, or a string of characters, typically containing a failure message. The "Right" and "Left" are just ways of "wrapping" the type, so that it is of type (Except α) rather than α or String.

"The present king of Springfield" can now be said to denote (Left "No such entity."), or whatever message be consider useful - it doesn't matter which. The verb "stopped-loving" (which for simplicity we treat as a fixed unit) has type (e $\rightarrow$ e $\rightarrow$ Except t). For instance, "Homer stopped loving Bart." might return (Left "He never did in the first place."), according to the model of section (1.1), while "Homer stopped loving Lisa." returns (Right True) and "Lisa stopped loving Bart." returns (Right False).

As before, we need to define composition of these types, so we define two functions, *return* and *bind*, of the same types as always (modulo the new type constructor).

bind x f =

- For x of the form (Left y), then: (Left y)

- For x of the form (Right y), then (f y)

As before, *return* and *bind* together provide all the compositionality we need for our new types. For example "Homer stopped loving Bart" can be calculated as

```
homerStoppedLovingBart = do
  x <- (return Homer)
  y <- (return Bart)
  return (stopped-loving y x)
```

Here, the order of binding only influences which cause of presupposition failure has its message reach the sentence level. As an example, we observe that the denotation of "Homer stopped loving Bart" will be (Left "He never loved him."), or whatever message we define, in our denotation for *stopped loving*.

Another word which triggers a presupposition is "both". We can define this as: both x = λy $\rightarrow$ (Right ($\forall$z (x z) $\rightarrow$ (y z))) if x = 2, else (Left "There are more than two.")

1.4 Conventional Implicature: The Writer Monad

Conventional implicatures, following (Potts 2005), can be thought of as another dimension of meaning, separate from the *at-issue* meaning, i.e. the main meaning of a proposition. For example, (8) conveys both that Homer runs and that he is a parent. However, the negation of (8) does not constitute a denial that

---

[5]Previous attempts to model presupposition failure monadically have used the Maybe monad; Dylan Bumford and Chris Barker allude to such a usage at `https://github.com/dylnb/esslli2015-monads/tree/master/intro`. We propose to use the Exception monad instead, so that strings containing information about the nature of the failure can percolate up the tree.

[6]Strictly speaking, Except is parametrized over any type β, so that values are either of the form (Left β) or (Right α). We just fix β to the type of character strings, since this is what we will use here.

Homer is a parent. In fact, the sentence "It is not the case that Homer, a parent, runs." still asserts that Homer is a parent.

(8)   Homer, a parent, runs.

There are many varieties of conventional implicature. Following (Giorgolo and Asudeh 2012), we just model appositives, as in (8). Once again, we do this with a type constructor, called Writer, where (Writer α) is a synonym for (α, (List t)), i.e. a pair of values, the first of type α, the second of type (List t), a list of truth values[7].

We term the value on the left of the tuple the at-issue meaning, and the value on the right the side-issue meaning. The idea is that certain expressions, like "Homer, a parent" may contain side-issue meanings which we wish to compose through the tree. Thus, we want (9) to have the denotation (False,[True,True]), where the left hand value, False, represents the at-issue meaning, and the two right-hand values in the list are the side-issue claims, that Homer is a parent, and Bart is a child. Because there are two side-issue meanings, and there could be any number in theory, we place them in a list. We treat "comma" as a lexical item, so that "Bart, a child" is effectively (comma (child, Bart)), i.e. the function "comma" applied to the predicate *child* and the entity *Bart*, and has a denotation of (Bart, [(child Bart)]), which reduces to (Bart, [True]).

(9)   Homer, a parent, loved Bart, a child.

As before, we only have to define *bind* and *return*:

- bind x f = (f (fst x), snd (f(fst x)) + (snd x)), where fst(x,y) = x, snd(x,y) = y, and (+) is list concatenation.

- return x = (x,[])

We define *comma*, of type ((e $\rightarrow$ t) $\rightarrow$ e $\rightarrow$ (e, list t)) as: comma x y = (y, [(x y)]). *Comma* takes a predicate and an entity, and returns the entity along with a list of just one side-issue meaning, stating that the predicate holds of the entity.

The Writer monad gives us the basic empirical properties we want: altering the at-issue content does not alter the side-issue content, and while a side-issue meaning can depend on an at-issue one, the reverse is not true. The following do-expression yields the value (False,[True,True]):

```
homerUsedToLoveBart = do
  x <- comma parent Homer
  y <- comma child Bart
  return (loved y x)
```


1.5   Intensionality: the Reader Monad

To handle intensionality, we introduce a type constructor called Reader. (Reader a) is a synonym for (s $\rightarrow$ α), where s is the type of worlds[8]. Reader thus takes a type α and gives us the type of a function from worlds to α.

Modal operators like *must* and *can* can be assigned the type (Reader t $\rightarrow$ Reader t), or even ((List w) $\rightarrow$ Reader t $\rightarrow$ Reader t) if be want to represent a Kratzer style conversational background. Entities and predicates can also be relativized to a world (following (Giorgolo and Asudeh 2014), so that *Homer* is of type (Reader e) and *child* is of type (Reader (e $\rightarrow$ t)). *Believe* is of type (Reader t $\rightarrow$ e $\rightarrow$ Reader t) and for arguments p and x, returns True at a world w iff p is true in all of the belief worlds of x.

We can now introduce *return* and *bind*:

---

[7]As with Except, we have simplified matters. In fact, Writer is parametrized over any monoidal type β for the second member of the tuple - we just fix it to (List t).

[8]In our Haskell implementation, we simply set s to Int, the type of integers.

- return x = λ y $\rightarrow$ x

- bind x f = λ y $\rightarrow$ f (x y).

This allows us to capture the difference between two readings of (10), the first where Homer believes of Lisa that she is Bart, and the second where he believes the proposition "Bart is Lisa"[9].

(10) Homer believes Bart is Lisa.

Suppose we define Lisa, of type (I e) as (λx $\rightarrow$ Bart if x = 1, else Lisa) and Bart as (λx $\rightarrow$ "Bart"). We then say that Homer's belief worlds are the singleton {1}, i.e. that Homer believes whatever is the case in world 1. Then in do-notation, the two readings of (10) amounts to the difference between:

```
homerBelievesBartIsLisa1 = do
  y <- Lisa
  believes (bind Lisa (lambda x -> return (x == y))) Homer
```

```
homerBelievesBartIsLisa2 =
  believes (bind (Bart (lambda a ->(bind Lisa (lambda b ->return (a == b))))) Homer
```

In our model, the first of these, at the real world, is False, and the second True.

1.6 Dynamic Semantics: The StateSet Monad

Two basic properties of file change semantics, as laid out in (Heim 1983) are the behaviour of existential quantifiers, and anaphoric pronouns. Consider (11):

(11) A parent walked in. They love Bart.

The first sentence of (11) is true just in case some parent walked in. The second seems to be able to have "They" refer to whichever parent was referred to in the first sentence. Naturally, the truth of the second sentence depends on the choice of parent.

To model this sort of anaphora, we introduce a type constructor StateSet[10]. This type constructor takes α to ((List e) $\rightarrow$ List (α,List e))[11].

- bind x f = λy $\rightarrow$ flatten([((f j) k) for (j,k) in p]), where p = (x y)

- return x = λy $\rightarrow$ [(x,y)]

Flatten is defined as with the List monad. The syntax "[(f x) for x in y]" is to be understood as: for each element x in y of type (List α, change the value of x to (f x), and return a new list of these updated values.

To show how this works semantically, consider "A parent walked in.". The idea is that "a parent" is an expression of type (StateSet ((e $\rightarrow$ t) $\rightarrow$ t)) which adds a new entity to the list. However, because the entity could be either Marge or Homer, what is returned is a list of two possibilities of type (((e $\rightarrow$ t) $\rightarrow$ t), [e]), one which adds Homer to the list in the second part of the tuple, and one which adds Marge. For a lengthier and clearer explanation, see (Charlow 2014). For now, it is perhaps easiest to just see the result of (11) in our grammar. To encompass both sentences, we use the monadic *next* operator, of type (M a $\rightarrow$ M b $\rightarrow$ M b) defined as:

(12) next x y = bind x (λz $\rightarrow$ y)

---

[9]There are more readings, and these can be captured in a similar way.

[10]This approach is adapted from (Charlow 2014. He in fact defines the StateSet monad using a monad transformer, the mechanism we shall discuss in section (2))

[11]We have fixed the type of the state to (List e), but it could be any monoidal type.

In other words, *next* computes the first argument, throws away its value, while keeping its monadic effect, and then outputs the result of the second argument. In the case of the state monad, we use this to run one sentence, thus updating the state, and then use this state as input to a second sentence, which we want to evaluate. This is how we derive a denotation for the second sentence of (11). First, each sentence separately, in do-notation.

```
aParentWalkedIn = do
  x <- a parent
  return (x walked-in)

theyLoveBart = do
  y <- they
  return (love Bart y)
```

The denotation of (next aParentWalkedIn theyLoveBart) is then a function of type ((List e) → (t, (List e)). Given the empty list of entities, it returns: [(True, [Marge]),(True, [Homer])]. Note that a false sentence can still change the state. For example "Homer loved a parent" might be false, but still allows "they" in "They ran." to refer to Marge. This is captured by our monad.

Discourse referents can also be accessed in the same sentences in which they are introduced, as we would want. For example:

```
do
  y <- a parent
  x <- they
  return (y (loved x))
```

Note that it is impossible to derive a meaning for "Herself$_1$ loved a parent$_1$." here, on account of the fact that "a parent" is a scope taker of type (StateSet (e → t) → t) and so must be the subject. Empirically this is an advantage, but only as a result of the fact that we have not yet explored scope taking.

1.7   Scope Taking: the Continuation Monad

Scope takers like *everyone* appear in the positions where objects of type e appear, but are of type ((e → t) → t). We can think of them as of type (C e), where C is a type constructor which for every type α, gives us a new type (C α), defined as: ((α → t) → t).

Once again, composition is the problem at hand. We have expressions of type (C a) and also expressions of type (α → C α), like *every* and *some*. What the continuation monad does is to compose these values of type (C α) in the appropriate way.
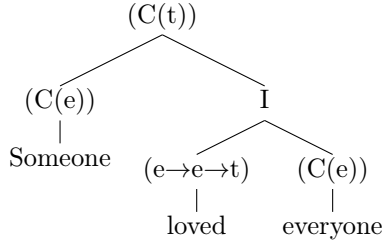
We define return x = λ y → y x.

We define bind x f = λc → x(λy → (f y) c)

With these, we can compute "Someone loved everyone", with the types as shown in the following tree, and values for *someone* and *everyone* as:

- everyone x = $\forall$y, x y
- someone x = $\exists$y, x y

(C(t))

(C(e))   I

Someone   (e→e→t)   (C(e))

loved   everyone

In do-notation, we compose as follows:

```
someonelovedEveryone1 = do
  x <- someone
  y <- everyone
  return (loved y x)
```

This gives us an linear scope reading. Recall that the order of binding, i.e. the order of the first two lines of the do-expression, did not matter in previous cases. Here it does. In fact, reversing their order gives us the inverse scope reading!

```
someonelovedEveryone2 = do
  y <- everyone
  x <- someone
  return (loved y x)
```

2 Combining Monads

With this introduction to our monads complete, we now turn to the main topic of the paper, their combination. It is natural to want a semantic grammar to handle more that one of the six phenomena above at once. A first attempt to do this would be to have multiple monads. Not only does this quickly become very complex, undermining the simplicity offered by the monadic approach, but it does nothing to model how the phenomena *interact*. The alternative is to construct a single monad which handles all of the effects we have discussed at the same time.

In Haskell, and functional programming more generally, the prevalent mode of composition of monads are known as monad transformers. These are higher order type constructors, which first take any monadic type constructor, and then return a new type constructor which is also a monad. For a number of monads, there is a corresponding monad transformer, which takes another monad and adds new functionality to it. For any monad M, here are the types of the six monad transformers corresponding to the six monads we have discussed[12]. To clarify, what is displayed below are the types we obtain when we supply a monad M to each monad transformer, and then supply a type α to the new type constructor we obtain:

- ListT: M $[\alpha]$
- StateSetT : (List e) $\to$ [M $(\alpha,$ (List e)]
- WriterT : M $(\alpha,$ (List t))
- ExceptT : M (Either String α)
- ReaderT : s $\to$ M α
- ContT : $(\alpha \to$ M t$) \to$ M t

---

[12]We preserve the simplifications we discussed for each monad. For example, WriterT is taken to add a list of truth values, rather than any monoidal type.

Because a monad transformer takes a monad and returns a monad, it is possible to stack them sequentially. For example, (ExceptT List) is a monad which in turn can be taken by (WriterT), to yield a new monad which handles both conventional implicature and presupposition failure. Our goal, then, will be to stack all seven monads, in order to produce a grammar which handles all their side effects and interactions at once.

To this end, we must determine the order to stack our seven monads. As it turns out, this is an empirical question, and requires us to consider linguistic data. To show this, let us consider, as an example, the two possible ways of combining List and Except. We could either have (ExceptT List) or (ListT Except). Following the above definitions, the latter gives us (Either String $[\alpha]$), (i.e. a list which could fail of values, of type α) and the former gives us [Either String α] (a list of values, each of which can fail). Which is appropriate for modeling natural language semantics? Take sentence (13):

(13) "Bart$_f$ stopped loving Lisa."

Recall our model, defined in (1.1). In particular:

- Bart$_f$: [Bart, Lisa, Homer]
- *loved* : {(Lisa, Homer), (Bart, Lisa), (Maggie, Homer)}
- *stopped loving* : {(Lisa, Homer), (Maggie, Homer)}

Since Homer never loved Lisa in the first place, and the proposition "Homer stopped loving Lisa" is in the set of focus alternatives for (13), should presupposition failure be triggered? If so, should the entire proposition fail, or just the focus alternative?

This is an empirical question about actual language. We judge that (13) would not incur a presupposition failure if it were the case that Homer were in the set of alternatives of "Bart$_f$" and had never loved Lisa in the first place. For this reason, we want failure to apply separately to each focus alternative, rather than collectively for all of them together. Thus, we have to choose (ExceptT List).

This is the sort of reasoning which will guide our choice of order of the monads. It may turn out that other monads intervene between ExceptT and List, for instance, for could have (ExceptT (WriterT List)). However, the above reasoning is still applicable, and demonstrates that ExceptT should be higher on the stack of transformers than List, regardless of what comes in between them.

At the risk of giving the game away, we can reveal that the order which we have found to produce the correct behavior for natural language semantics (at least in English) is as follows:

(14) ContT >ReaderT >ExceptT >WriterT >StateSetT >List

In the following section we justify this claim, by considering many of the pairs of monads - there are fifteen pairs in total - and discussing the main empirical data regarding their interaction. We then ascertain in which order each pair should apply, in the manner exemplified above. As we shall see, many of the interactions between monads relate to familiar problems in natural language semantics.

2.0.1 List and StateSet: Focus and Dynamics

How should focus interact with dynamic semantics? Consider (15)

(15) Bart$_f$ loved a parent.
(16) They ran.

As discussed in section (1.6), "a parent" creates a discourse reference. However, we do not want the focus alternative sentence to also create a discourse reference. For example, (16) could not refer to the parent that Lisa loved, supposing that Lisa was the focus alternative to Bart. For this reason, StateSet must scope

outside of List. Otherwise, List would distribute over StateSet, and focus alternatives would multiply the discourse referents created.

The type signature of our new monad looks as follows, where α is the type:

(17) $((\text{List e}) \rightarrow \text{List (List (} \alpha,(\text{List e})))$

Monad transformers are equipped with a function *lift* in Haskell, which raises an element from type (M α) to type (T M α) for any monad and monad transformer M and T. This makes it easy to add monad transformers to our grammar without redefining everything else. For reasons of simplicity, we have omitted *lift* from our calculations in the rest of the paper.

Calculating the result of (19), using the state produced by (18), for the domain defined in section (1.1), gives us a value of: [[(True,["Bart"]), (True, ["Lisa"]), (False,["Maggie"])], [(True,["Bart"]), (True, ["Lisa"]), (False, ["Maggie"])], [(True, ["Bart"]), (True, ["Lisa"]), (False, ["Maggie"])]].

(18)   A parent ran.

(19)   They love Bart$_f$.

What does this mean? It is a set of three focus alternatives, each of which consists of a list of tuples of a truth value and a list of entities, the possible discourse referents. This complexity may seem overwhelming, but it precisely the reason we need to use a monad. Understanding how such complex types compose is otherwise impossible. Note that our system produces a list of focus alternatives for "They ran" when it is run after a sentence with focus alternatives. This seems unnecessary at best, and a problem at worst for our system.

2.0.2   Writer and StateSet: Conventional Implicature and Dynamics

To determine the order of these monads, consider:

- Bart, who a parent loved, ran.

(2.0.2) is an example of the form discussed in (Giorgolo and Asudeh 2012) as a potential objection to Potts' rule of no information flow from CI to at-issue meaning. Asudeh accounts for similar problems using monad transformers, though this particular case is not treated.

(2.0.2) provides us, therefore, with a justification for having Writer scope outside StateSet. We *do* want the effects of side-issue meaning to affect the state. We therefore add Writer to our current stack. Our new monad has the following type signature:

(20) $((\text{List e}) \rightarrow \text{List (List (} \alpha,(\text{List t}),(\text{List e})))$

The following do-expression calculates the denotation of (2.0.2):

```
bartWhoKnowsASingerLikesThem = do
  x <- a parent
  y <- comma (lambda f -> x (loved f)) Bart
  return (ran y)
```

Its value is: [[((True, [False]), ["Homer"]), ((True, [False]), ["Marge"])]]. The outer brackets represent the fact that there are no focus alternatives barring the actual sentence. For each possible assignment of a referent to "a parent" we have a different output sentence and state with a different discourse referent. In each case, the side-issue meaning is a single false claim, since neither parent loved Bart, and the at-issue meaning is true.

2.0.3 Writer and List: Conventional Implicature and Focus

With our current stack of monads, we can consider one further interaction that arises, that between Writer and List. For example:

(21)   Bart, a $child_f$, ran.

Our calculation is similar to before. We take $parent_f$ to denote $(\lambda x \rightarrow [\text{parent x, child x}])$:

```
bartAChildRan = do
  x <- lift parentF
  y <- commaW x "Bart"
  return $ ran y
```

The resultant value is [[((True,[False]),[])],[((True,[True]),[])]], which contains two focus alternatives, which have escaped from the side-issue meaning.

2.0.4 Except and Writer: Presupposition Failure and Conventional Implicature

The key data in this case is the following:

(22)   Bart, who both Simpsons love, ran.

(23)   Both of the children love Homer, a Simpson.

(22) demonstrates that something inside a conventional implicature can trigger presupposition failure. This suggests that ExceptT must be above WriterT in the stack.

In the second case, (23), we observe that a presupposition failure in the main meaning does not prevent a conventional implicature being generated, that Homer is a Simpson. This shows that our type must restrict failure to the at-issue meaning, which is what scoping ExceptT on the outside achieves.

Our new monad has the following type signature:

(24) $((\text{List e}) \rightarrow \text{List (List (Either String } \alpha,(\text{List t}),(\text{List e})))$

We can then compute the value of (22), with the following do-expression:

```
bartWhoBothSimpsonsLoveRan = do
  x <- both isSimpson
  y <- comma (lambda f -> x (loved f)) Bart
  return (ran y)
```

This gives a state change function. Run on the empty list, it returns: [[((Left "There are more than two.",[]),[])]]. The is a singleton - there are no focus alternatives. The at-issue meaning is a presupposition failure, i.e. a value of the type (Left α). Its side-issue meaning list is empty, as is the list of discourse referents. This seems empirically correct. Of particular interest is the fact that the main meaning is caused to fail by information in the side-issue meaning.

For (23), we use the following do-expression:

```
bothOfTheChildrenLoveHomerASimpson = do
  y <- comma isSimpson Homer
  x <- both child
  return (x (loved y))
```

This gives [[((Left "There are more than two.",[True]),[])]]. Again, there are no focus alternatives or discourse referents. But crucially, the side-issue meaning has survived the presupposition failure, precisely the empirical result we wanted to account for.

Interestingly, the side-issue meaning does not survive if we change the order of binding, i.e. the order of the middle two lines of the do-expression. In this version, the presupposition failure happens before the side-issue meaning is created. Whether this is a boon is unclear; are there situations in which we want the side-issue meaning to not be generated?

2.0.5 List and Except: Focus and Presupposition Failure

Our monad, as it now stands, is capable of representing two more interactions not yet discussed, the first is between List and Except, and the second between List and StateSet. For the first, consider:

(25) Homer$_f$ stopped running.

(26) Both children$_f$ ran.

Does (26) incur failure for all of the focus alternatives, supposing that "Bart stopped running" is one of them? Our theory predicts that it does not. This seems to be a positive consequence; we could say "Only Homer stopped running" and avoid a presupposition failure. (Note that this would require an account of "only", which we have not discussed in this paper.)

2.0.6 Reader and Except: Intensionality and Presupposition Failure

The most obvious interaction we want to model is the fact that (27) can avoid presupposition failure even if Homer never ran in the first place, since Lisa may believe either that Homer is actually someone else, or that Homer did in fact run.

(27) Marge believes Bart stopped running.

ReaderT is commutative, in the sense that for another transformer T, the order (ReaderT >T) yields the same type as (T >ReaderT). For this reason, it is unclear what the effect of changing its position in the stack is.

We can now add ReaderT to our stack, giving us the type constructor:

(28) w → ((List e) → List (List (Either String α,(List t),(List e)))

Sentence (27), under the reading where "Bart" scopes inside "believe", can now be expressed as "margeBelievesBartStoppedRunning" in the following:

```
bartStoppedRunning = do
  x <- bart
  stoppedRunning x

margeBelievesBartStoppedRunning = believe bartStoppedRunning Homer
```

Supposing that in Marge's belief worlds, Bart is actually Homer, this avoids presupposition failure, returning [[((Right True,[]),[])]].

2.0.7 Reader and Writer: Intensionality and Conventional Implicature

A key observation of (Potts 2005) is that CIs are speaker independent. For instance, in (29), the fact that Bart is a child does not depend on Homer.

(29)   Homer believed he saw Bart, a child.

Our system has no built in mechanism to prevent the predicate in an apposition from being interpreted relative to the speaker. Such readings are perfectly possible here. Given the nuance of the issue, discussed at greater length in (Harris and Potts 2009), this may not be a bad thing.

2.0.8 Cont and Reader: Scope and Intensionality

The continuation monad transformer is the last, and most problematic member of our stack. As noted in (Shan 2002), it seems like we would want the type of "everyone" in (30) to be ((Reader e $\rightarrow$ Reader t) $\rightarrow$ Reader t). This is because "believes some child ran" is of type (Reader e $\rightarrow$ Reader t). However using the continuation monad gives us the type ((e $\rightarrow$ Reader t) $\rightarrow$ Reader t).

(30)   Every parent believes some child ran.

The extent to which this is a problem in practice is a topic for further study. It is certainly the case, however, that Cont must scope at the very top, if only for technical reasons. Suppose we stack Except on top of it, for instance. We would then obtain the type constructor ((Except $\alpha \rightarrow \alpha) \rightarrow \alpha$). Suppose further that we perform monadic composition, to end up with a value of type ((Except t $\rightarrow$ t) $\rightarrow$ t). We must then feed this a function of type (Except t $\rightarrow$ t) in order to receive a value we can access. Since no such function is generally defined, this cannot be the correct order in the stack. Similarly, *mutatis mutandis* for the other monads.

Adding our final monad transformer gives us a now formidable looking type constructor:

(31) $(\alpha \rightarrow$ (w $\rightarrow$ ((List e) $\rightarrow$ List (List (Either String α,(List t),(List e)))) $\rightarrow$ (w $\rightarrow$ ((List e) $\rightarrow$ List (List (Either String α,(List t),(List e))))

But we need not worry about keeping track of the type in our computations. The whole point is that (31) is a monad, so all the complexities of its composition are handled by *bind* and *return*.

2.0.9 Cont and Except: Scope and Presupposition Failure

Interactions of effects arise between *Cont* and *Except* when we are dealing with quantification and failure at the same time. The following sentences illustrate some potential such interactions:

(32)   Every child stopped loving some parent.
(33)   Some parent loved both children.

In (32), two scope readings are available. With linear scope, the sentence should be false, under our model of section (1.1). However, with inverse scope, it should trigger presupposition failure, since it presupposes that there is a unique parent that each child loved. In (33), the sentence should also fail in our model, since *both* presupposes that there are two children. It should fail whichever scope reading we choose.

Despite the problems with ContT discussed in (2.0.8), our model handles the basic interaction with presupposition failure. It calculates the value of (33) as [[((Left "presupposition failure: they never loved them",[]),[])]]. However, it is not capable of capturing the difference between the two readings of (32)[13]

---

[13]This is due to a technical difficulty, where any presupposition failures in the whole domain of quantification for existentials propagate through the composition. Again, this raises the question of whether existentials should be scope takers.

3   Conclusions

This brief survey hardly does justice to the complexity of the interactions of these phenomena and monads. Particularly for intensionality and scope, a huge amount can and has been said which we have yet to adapt to our theoretic perspective. For example, elements which are both scope taking and intensional, like modals, seem ideal for a treatment in this framework.

There also seem to be fundamental reasons why the continuation monad is too weak to handle the correct interaction of scope with presupposition failure, discourse referents and intensionality.

Furthermore, we have omitted discussion of the interactions of several pairs of monads. For instance, focus and scope taking, and intensionality and state are combined in:

- Everyone loved $someone_f$.
- Bart believes a parent loved him. They ran.

Both of these seem to involve very subtle interactions, about which little has been said in the literature. Furthermore, we have only considered interactions between *pairs* of effects. The real power of the mechanism introduced in this paper is to handle the interaction of many, even all six, of the phenomena under consideration. In some cases, this works well. For instance, consider:

(34)   They, a $parent_f$, stopped loving every parent.

Though this is obviously a somewhat contrived example, our grammar is capable of performing composition in the right way, through the following do-expression:

```
theyAParentStoppedLovingEveryParent = do
  x <- they
  a <- parentF
  w <- comma a x
  z <- every parent
  stoppedLoving z w
```

If we lower the continuation, and run the resulting function on a list of just one discourse referent, ["Homer"], we obtain:

(35)   [((Right False, [True)
           ,["Homer"])], [((Right False, [False]), ["Homer"])]]

This contains two focus values, one for "Homer, a parent, stopped loving every parent." and one for "Homer, a child, stopped loving every parent.". In each case, no presupposition failure has been triggered, since Homer did indeed once love every parent. In the first focus alternative, the one side-issue meaning is true, and in the second, it is false. The discourse referent, "Homer", has been passed on in either case.

That the grammar is able to handle focus, state, conventional implicature, presupposition failure and scope taking at once is a testament to the technical power of this approach.

3.1   Applicatives

In this paper, we have defined monads for each type constructor in question. These assign functions *bind* and *return* to the type constructor in question. A weaker construction is the applicative, which instead of *bind*, has a function *apply* of type $(\alpha \rightarrow M\ (\alpha \rightarrow \beta) \rightarrow \beta)$, for a given type constructor M. This is strictly weaker than a monad, in the sense that a *apply* can be defined in terms of *bind*, but not vice versa.

However, applicatives can be composed much more straightforwardly. For many of the phenomena we have discussed, applicatives suffice in place of monads. The question, therefore, is whether they could provide a more workable alternative to a monadic grammar. For more on this possibility, see (Kiselyov 2015).

3.2

This paper has covered a wide range of material in a very short space. We would like to highlight three main conclusions:

- Our grammar replicates the results of (Giorgolo and Asudeh 2012) and generalizes the concept of using monad transformers to model the interaction of effects. As (34) demonstrates, it is capable of modeling the behavior of multiple phenomena at once.

- The order of the stacked monads is crucial to ensuring the empirically correct behavior for our semantic grammar.

- Haskell is perfectly adapted to designing semantic grammars which are transparent enough to have denotational utility, and very concise. The code evidencing the findings of this paper is available online at `https://github.com/reubenharry/reubenharry.github.io/tree/master/grammar`.

The key areas of difficulty that merit further study are the correct use of the Continuation and Reader monad transformers, and the interaction of focus (the List monad) with the other phenomena under consideration.

References

Barker, Chris (2002). "Continuations and the nature of quantification". In: *Natural language semantics* 10.3, pp. 211–242.

Charlow, Simon (2014). "On the semantics of exceptional scope". PhD thesis. New York University.

Giorgolo, Gianluca and Ash Asudeh (2012). "M, η,*> Monads for conventional implicatures". In: *Proceedings of sinn und bedeutung*. Vol. 16, pp. 265–278.

— (2014). "Monads as a solution for generalized opacity". In: *EACL 2014*, p. 19.

Harris, Jesse A and Christopher Potts (2009). "Perspective-shifting with appositives and expressives". In: *Linguistics and Philosophy* 32.6, pp. 523–552.

Heim, Irene (1983). "File change semantics and the familiarity theory of definiteness". In:

Kiselyov, Oleg (2015). "Applicative abstract categorial grammars". In: *Proceedings of the Third Workshop on Natural Language and Computer Science.*

Potts, Christopher (2005). *The logic of conventional implicatures*. Vol. 7. Oxford University Press Oxford.

Shan, Chung-chieh (2002). "Monads for natural language semantics". In: *arXiv preprint cs/0205026*.