VP Ellipsis
COSI 135 Computational Semantics
Final Project

Hayley Ross

December 13, 2019

1 Introduction

In this project, we will explore how to model VP ellipsis using a continuation-based approach to anaphora, modelled in λ-calculus and implemented in Haskell. We follow Van Eijck and Unger (2010) in their general approach towards handling context with continuations, and draw on De Groote (2006), Barker and Shan (2014) and Asher and Pogodalla (2010) for details of the continuized meanings of our constituents. This report accompanies an implementation in Haskell, which can be viewed at `https://github.com/rossh2/cosi135-vp-ellipsis`.

We begin in Section 2 by introducing the phenomenon of VP ellipsis and laying out how De Groote and Barker and Shan use continuations to handle pronominal anaphora, which will form the basis for our treatment of VP ellipsis. Then, in Section 3 we set up an extension of this model which handles VP ellipsis with simple predicates such as *laughed*. We then discuss it, and revise it to handle unsaturated predicates such as *loved his sword*. (This example may seem an odd one to pick a priori, but we must work within the constraints of our toy grammar.)

In Section 4, we explore handling VP ellipsis constructions with varying tense and aspect. We extend both the grammar and the model to handle a variety of English tenses, as well as the perfect aspect. Finally in Section 5, we discuss a number of ways in which the model could be extended, such as handling more complicated pronominal anaphora outside and inside VP ellipsis, handling subtler time references in VP ellipsis, and the double VP ellipsis which can occur in conditionals.

2 Background

2.1 VP Ellipsis

VP ellipsis is a quintessential context-dependency phenomenon. It captures fact that when we want to repeat a previously mentioned event, we need not spell out the same words again, we can instead refer to it simply with *did too*:

(1) Dorothy laughed. Alice did too.

(2) Alice helped Atreyu. Dorothy did too.

(3) Dorothy has defeated a wizard. Alice has too.

(4) Alice cheers. Dorothy will too.

(5) Atreyu loves his sword. Little Mook does too.

Because certain actions are already given by the context, we can replace them with the simple *did (has, will) too* when we want to repeat them. In other words, VP ellipsis provides an excellent way to study what is (or should be) in the context: if we can omit it, the context must contain it. There are other kinds of ellipsis such as NP ellipsis, as well as other kinds of anaphora such as pronominal anaphora, all of which provide further 'windows', to use Cann, Kempson, and Gregoromichelaki (2009)'s terminology, on what is in the context. Here, we focus on VP ellipsis because it shows that the context does not just contains objects from the model such as entities or predicates; instead, it must contain at least some semantic representations.

VP ellipsis requires both syntactic and semantic analysis to explain all its behaviours; Cann, Kempson, and Gregoromichelaki (2009) gives an overview. For the purpose of this project, we will not engage with its syntactic puzzles, but rather concentrate on interpreting its meaning in discourse and its contribution to the contents of the context, using a simple grammar.

As shown by the examples, VP ellipsis does not simply "copy and paste" the meaning of the previous VP into the next: the theme of the verb may change. The wizard that Dorothy defeated need not be the wizard that Alice defeated. Little Mook most likely loves his own sword, not Atreyu's. (Both readings are however possible.) Further, the tense can change: the ellipsis sentence contains an auxiliary which carries the new time at which the event happens. Alice is cheering now but Dorothy will only cheer later. (Perhaps they are supporting opposing teams.) In this project, we will take apart these cases one by one to determine exactly what we must store in the context.

2.2 Grammar and Model

We will start with the following toy grammar, following Chapter 4 of Van Eijck and Unger (2010). Since this is such a small model, we will include the lexicon directly in the grammar, as is common in computational approaches. Our lexical items have no features such as gender or singular/plural, they are just strings. (As we will see later, this will simplify our selection function for pronouns but also cause some minor problems when implementing tenses, as our grammar will not be able to distinguish certain cases by features.)

S → NP VP
NP → DET CN
NP → *Snow White* | *Alice* | *Dorothy* | *Goldilocks* | *Little Mook* | *Atreyu*
NP → *everyone* | *someone*
NP → *he* | *she* | *it*
DET → *every* | *some*
CN → *girl* | *boy* | *princess* | *dwarf* | *giant* | *wizard* | *sword* | *dagger*
VP → TV NP
VP → *laughed* | *cheered* | *shuddered*
TV → *loved* | *admired* | *helped* | *defeated*

(Note this omits the ditransitive verb and 'NP → NP *that* NP TV' constructions from Van Eijck and Unger since they do not relate to the problem of VP ellipsis; we can model all the phenomena we need without them.)

This grammar is easily implemented in Haskell by defining S, NP etc. as `data` types; see Van Eijck and Unger (Chapter 4) or the Haskell implentation accompanying this report. The grammar corresponds to a model, also following Van Eijck and Unger (Chapter 6.3). The model holds a list of entities as well as

functions for each of the one-place and two-place predicates above which map entities or pairs of entities to booleans.

Some technical detail is required to set this up. If we define our entities as a set of objects A-Z, we can then map named entities and common nouns onto them. Recall that a common noun such as *girl* has meaning $\lambda x.\,\text{girl}(x)$ which translates directly into a lambda expression in Haskell, except that girl(x) checks against a list in Haskell.

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z

snowWhite = S
alice     = A
dorothy   = D
goldilocks = G
littleMook = M
atreyu    = Y

type OnePlacePred = Entity -> Bool

girl, princess, wizard :: OnePlacePred
girl     = \x -> x `elem` [S,A,D,G]
princess = \x -> x `elem` [E]
wizard   = \x -> x `elem` [W,V]
...
```

Similarly, we can define our verbs:

```
laugh, cheer, shudder :: OnePlacePred
laugh   = \x -> x `elem` [A,G,E,W]
cheer   = \x -> x `elem` [M,D,A]
shudder = \x -> x `elem` [S,Y]

type TwoPlacePred = Entity -> Entity -> Bool

love :: TwoPlacePred
love = \x y -> (x,y) `elem` [(Y,E),(B,S),(R,S),(F,Y),(C,M),(X,E)]
...
```

(Some helper functions have been omitted in these snippets to make the code easier to follow.)

2.3 Continuation Passing Style as a Model for Context

Continuations are a style of functional programming in computer science which abstract over the current computation, and add the future computation as an extra parameter. We can use this in linguistics to abstract over the current proposition in a discourse, and allow it to consider its left context (previous propositions) and its right context: the future propositions, i.e. future computation of the discourse meaning. In essence, we already consider the future computation when we use generalized quantifiers in semantics, since we talk about all the properties that hold for Dorothy (that she is singing, that she

wears a dress, and so forth), in other words the things that she could be doing in the VP of the sentence, instead of just talking about Dorothy as an entity. Continuation passing style takes this a step further, and applies this to all parts of the sentence: each defines their meaning in terms of their continuation, namely what happens around them. A detailed motivation of continuations and how they relate to generalized quantifiers can be found in Van Eijck and Unger (2010), Chapter 11.

This section details how to add left and right contexts to an otherwise traditional semantic model in λ-calculus. In principle, we are using a DRT/SDRT framework to handle discourse, as Asher and Pogodalla (2010) and Barker and Shan (2014) do, but since our discourses are so short and we do not introduce new entities in them, this will not show through a great deal. No prior knowledge of DRT or SDRT is required to follow this report.

Following Van Eijck and Unger (2010), Chapter 12, we take the type of left contexts to be stacks (technically, lists) of entities and right contexts to be continuations: a discourse that requires a left context to yield a truth value. That is, left contexts have type `Context = [Entity]` and right contexts have type `Context -> Bool`, and the type of a sentence is `Sentence = Context -> (Context -> Bool) -> Bool`. Since discourses are formed by composition of sentences, we take the type of a discourse to be the same as the type of a sentence.

Following De Groote (2006), nouns have continuized type `Noun = Entity -> Sentence` and noun phrases have type `NounPhrase = (Entity -> Sentence) -> Sentence`. So, following Barker and Shan (2014), we lift an entity such as `snowWhite` to an NP with

$$[\![\text{snowWhite}]\!] = \lambda P.\, \lambda i.\, \lambda k.\, P(\text{snowWhite})(\text{snowWhite} :: i)(k)$$

where P denotes a predicate, i is a left context, k a right context and snowWhite $:: i$ denotes adding the entity snowWhite to the top of the context i (which we defined as a stack of entities). This is reminiscent of a generalized quantifier interpretation of nouns, where $[\![\text{snowWhite}]\!] = \lambda P.\, P(\text{snowWhite})$. Here, we also have the two contexts, and we will see that we can lift not only nouns but also to VPs, determiners and every other part of a sentence in this way.

In Haskell, this trivially becomes

```
liftNP :: Entity -> NounPhrase
liftNP e = \p i k -> (p e) (e:i) k
```

Similarly, De Groote (2006) gives the interpretation of common nouns as

$$[\![\text{girl}]\!] = \lambda x.\, \lambda i.\, \lambda k.\, \text{girl}(x) \wedge k(i)$$

where $\text{girl}(x)$ is the non-continuized meaning of *girl* in our model. These have type `Entity -> Sentence`, the definition of `Noun`. As for proper nouns, we can implement this and the following λ-expressions directly in Haskell using its λ-expression syntax – see the accompanying code for details.

VPs have type `NounPhrase -> Sentence` and so we define representations for intransitive and transitive verbs, following the meanings given in Barker and Shan (2014). (In Haskell, these are implemented as `liftVP` and `liftTV`.) Note that while these and common nouns are both predicates in our model, they have different types and so their representations differ accordingly.

$$[\![\text{cheer}]\!] = \lambda P.\, P(\lambda x.\, \lambda i.\, \lambda k.\, \text{cheer}(x) \wedge k(i))$$
$$[\![\text{love}]\!] = \lambda P.\, \lambda Q.\, Q(\lambda x.\, P(\lambda y.\, \lambda i.\, \lambda k.\, \text{love}(y)(x) \wedge k(i)))$$

All that remains is to define the internal structure of NPs. To begin with, we will only consider two types of determiner to go with our common nouns: *some* and *every*, which correspond exactly to the

predicate logic quantifiers $\exists$ and $\forall$, and to the Haskell functions `any` and `all`. Later, we will extend this to possessives *his, her, its*, but we will not implement *the* or *a* here: they raise many questions of their own regarding what entities they refer to and introduce in the context (such as whether the uniqueness of *the* must hold in the whole model or just in the context). These questions are mostly orthogonal to our study of VP ellipsis (one exception is discussed in Section 5.4) and so we will set them aside here, though of course to implement natural-sounding sentences, a good model should handle them.

Our two determiners *every* and *some* have continuized type `Noun -> NounPhrase`. De Groote (2006) provides the following continuized meanings for *some*:

$$[\![\text{some}]\!] = \lambda P.\, \lambda Q.\, \lambda i.\, \lambda k.\, \exists x P(x)(i)(\lambda i'.\, Q(x)(x :: i')(k))$$

Instead of checking $\exists x P(x) \wedge Q(x)$ as we would in the non-continuized version, we now check $P(x)$ with $Q(x)$ in its right context.

For *every*, the non-continuized meaning is $\lambda P.\, \lambda Q.\, \forall x P(x) \rightarrow Q(x)$ which, converting $\rightarrow$ to $\neg$ and $\wedge$, is equivalent to $\lambda P.\, \lambda Q.\, \forall x \neg (P(x) \wedge \neg Q(x))$. This matches the operations in De Groote's continuized meaning:

$$[\![\text{every}]\!] = \lambda P.\, \lambda Q.\, \lambda i.\, \lambda k.\, (\forall x \neg (P(x)(i)(\lambda i'.\, \neg Q(x)(x :: i')(\lambda i''.\, \top)))) \wedge k(i)$$

Here, $\lambda i''.\, \top$ is the trivial continuation. De Groote explains that this "allows DRT accessibility constraints to be satisfied by limiting the scope of the universal quantification."

Finally, the meaning of a pronoun such as *she*, following De Groote (2006), is

$$[\![\text{she}]\!] = \lambda P.\, \lambda i.\, \lambda k.\, P(\texttt{sel}(i))(i)(k)$$

where $\texttt{sel}(i)$ is a selection function that selects the discourse antecedent inside the context i. As Asher and Pogodalla (2010) point out, it should select a *suitable* antecedent, but for simplicity we will have it always select the entity at the top of the stack, following Van Eijck and Unger (2010). Thus, it will ignore features such as gender, so the implementations of *he*, *she* and *it* will be identical. We will discuss extensively in Section 5 how this could be improved.

How do we assemble these meanings into sentences? We saw that VPs have type `NounPhrase -> Sentence`, in other words, a VP is a function that takes an NP. Likewise our determiners take a (common) noun as input. Starting from the top, we can compose a sentence with an existing discourse as follows (following De Groote (2006) again):

$$[\![D.S]\!] = \lambda i.\, \lambda k.\, [\![D]\!](i)(\lambda i'.\, [\![S]\!](i')(k))$$
$$[\![S = \text{NP VP}]\!] = ([\![\text{VP}]\!][\![\text{NP}]\!])([\,])(\lambda i''.\, \top)$$

where $[\,]$ is the empty context and $\lambda i''.\, \top$ is the trivial continuation, as seen in $[\![every]\!]$.

If the NP is a proper noun, we go straight to the proper noun case above, otherwise we have

$$[\![\text{NP} = \text{DET CN}]\!] = [\![\text{DET}]\!]([\![\text{CN}]\!])$$

Likewise if the VP is intransitive, we go straight to the formula above (`liftVP` in Haskell), otherwise, we split it in a similar fashion:

$$[\![\text{VP} = \text{TV NP}]\!] = [\![\text{TV}]\!]([\![\text{NP}]\!])$$

Details of exactly how these all compose and how we map the syntactic forms declared in our grammar to the model predicates and constants can be seen in the Haskell implementation.

As desired, this structure allows us to model pronominal anaphora in sentences such as

(6) Alice laughed. She cheered.

(7) Some wizard shuddered. He laughed.

(8) Atreyu defeated some wizard. He laughed.

With our selection function definition, (8) is interpreted as *the wizard laughed*, which is one of the two possible interpretations (the other being that Atreyu laughed). Both interpretations are equally plausible without further context, so this is satisfactory, provided that we want our model to produce a single decisive interpretation rather than all possible interpretations.

With all this in place, we can now move on to extending the model to handle VP ellipsis, where we will see that we can use the same structure for resolving pronoun anaphora in the context to interpret VP ellipsis.

3 Handling VP Ellipsis

3.1 A Naive Implementation

We begin by extending our grammar with the following new rules:

VP $\rightarrow$ AUX Too
AUX $\rightarrow$ *did*
Too $\rightarrow$ *too*

This allows us to generate the second part of a VP ellipsis: *[. . .] Alice did too.*

How do we extend our continuation-based theory to handle this? A simple way is to reason as follows: our context current contains entities to handle 'missing' NPs (represented by pronouns). Entities are part of our model. Now that we need to handle missing VPs, we can add predicates from our model to the context; specifically, one-place predicates, which await a subject. If the VP ellipsis refers to a VP with a two-place predicate that has a complement, we can simply bind that complement to the two-place predicate to create a one-place predicate out of it (also known as currying).

So we redefine `data Context = Context [Entity] [OnePlacePred]`. We then modify our representations for intransitive and transitive verbs (`liftVP` and `liftTV`) so that they add to the context when they process a predicate (using the notation :: flexibly for adding both entities and predicates to the context):

$$[\![\text{cheer}]\!] = \lambda P.\, P(\lambda x.\, \lambda i.\, \lambda k.\, \text{cheer}(x) \wedge k(iv :: i))$$
$$[\![\text{love}]\!] = \lambda P.\, \lambda Q.\, Q(\lambda x.\, P(\lambda y.\, \lambda i.\, \lambda k.\, \text{love}(y)(x) \wedge k((tv(y)) :: i)))$$

Notice how $(tv(y)) :: i)$ binds the entity y, which is supplied to the complement NP, to the transitive verb when it adds it to the context. That is, the interpretation of y is fixed here.

Now, we can add another selection function, `selPred` which, exactly like `sel` for entities, just takes the predicate at the top of the stack, and use this to interpret 'did too':

$$[\![\text{did too}]\!] = \lambda P.\, P(\lambda x.\, \lambda i.\, \lambda k.\, \texttt{selPred}(i)(x) \wedge k(i))$$

The implementation in Haskell is virtually identical to the λ-calculus, albeit with some helper functions for readability – see the accompanying code for details.

This simple adaptation allows us to correctly interpret sentences such as:

(9) Alice cheered. Little Mook did too. [True in the model]

(10) Alice laughed. Little Mook did too. [False in the model]

(11) Goldilocks admired Dorothy. Atreyu did too. [True]

(12) Goldilocks admired some girl. Atreyu did too. [True]

(13) Some princess laughed. Some dwarf did too. [True]

3.2 The Need for Representations: Unsaturated Predicates

However, this implementation is naïve because it cannot handle cases where the predicate in the VP ellipsis is unsaturated. In those cases, the original VP contains a pronoun or other anaphor which needs to be resolved from the context:

(14) Atreyu raised his hand. Dorothy did too. (= Dorothy raised her hand.)

(15) Atreyu loved his sword. Little Mook did too. (= Little Mook loved Little Mook's own sword.)

(16) Dorothy visited her mother. Atreyu did too. (Who did Atreyu visit? Dorothy's mother or his own mother?)

Looking at our implementation carefully, we see that the complement of the predicate is fixed at the time when it is added to the stack; that is, any anaphora are resolved using the context *at that time*. Effectively, we store the previous context in our context. So in our current implementation, Atreyu visits Dorothy's mother, Little Mook loves Atreyu's sword, and Dorothy raises Atreyu's hand. While this is a possible reading, and sounds quite plausible for (16), for (14) this reading sounds very unlikely. This also is not the most common reading in any of the cases. We would prefer our model to choose the more common case, and resolve the pronoun in the new context where the possessive can refer to the new subject.

Instead of storing predicates in our context, we need to store VPs. Given the name 'VP ellipsis' for the phenomenon, this may seem obvious. However, previously our context consisted of objects from our model. Now we need to add VP representations, which are one stage away from being interpreted in the model. They have type `NounPhrase -> Sentence`. This allows us to interpret them in the new context when we encounter a VP ellipsis, and resolve anaphora such as possessive pronouns anew at that point.

Thus, our context has type

```
data Context = Context {contextEntities :: [Entity], vps :: [NounPhrase -> Sentence]}
```

As discussed, we need to add VPs to the context before we interpret them, so our intransitive and transitive verb representations (`liftVP` and `liftTV` functions) don't interact with the context at all: unlike in our naïve implementation, they can stay in their original state as taken from Barker and Shan (2014).

$$[\![\text{cheer}]\!] = \lambda P.\, P(\lambda x.\, \lambda i.\, \lambda k.\, \text{cheered}(x) \wedge k(i))$$
$$[\![\text{love}]\!] = \lambda P.\, \lambda Q.\, Q(\lambda x.\, P(\lambda y.\, \lambda i.\, \lambda k.\, \text{love}(y)(x) \wedge k(i)))$$

Instead, we change the representation of the VP. In Haskell, this is the function `intVP :: VP -> NounPhrase -> Sent` which calls `liftVP` and `liftTV`. (Note that `VP` is the type associated with the syntactic VP, not its representation in the model.) Previously, this function just mapped the syntactic predicates to the model

predicates and called `liftVP` and `liftTV`:

$$\texttt{intVP}(\text{cheered}) = \texttt{liftVP}(\text{cheer}) = [\![\text{cheer}]\!]$$
$$\texttt{intVP}(\text{loved } np) = (\texttt{intTV}(\text{love}))$$

Now, we specify the λ-bindings that this function has so that we can modify i as it passes through:

$$\texttt{intVP}(\text{cheered}) = \lambda P.\, \lambda i.\, \lambda k.\, (\texttt{liftVP}(\text{cheer}))P((\texttt{liftVP}(\text{cheer})) :: i)k$$
$$\texttt{intVP}(\text{loved } np) = \lambda P.\, \lambda i.\, \lambda k.\, ((\texttt{liftTV}(\text{love}))(\texttt{intNP}(np)))P(((\texttt{liftTV}(\text{love}))(\texttt{intNP}(np))) :: i)k$$

In fact, it is useful to define a helper function here, as it helps shows that the structure of the transitive and intransitive cases are very similar. Showing the implementation in Haskell,

```
liftAndAdd :: (NounPhrase -> Sentence) -> NounPhrase -> Sentence
liftAndAdd boundLift = \np i k -> boundLift np (addVP boundLift i) k

liftAndAddOnePlace :: OnePlacePred -> NounPhrase -> Sentence
liftAndAddOnePlace v = liftAndAdd (\t -> liftVP v t)

liftAndAddTwoPlace :: TwoPlacePred -> NP -> NounPhrase -> Sentence
liftAndAddTwoPlace tv objNP = liftAndAdd (\t -> (liftTV tv t) (intNP objNP))

intVP :: VP -> NounPhrase -> Sentence
intVP Laughed         = liftAndAddOnePlace laugh
intVP (VP1 Loved np) = liftAndAddTwoPlace love np
```

Our selection function `selVP` still just takes the head of the list (stack) of VPs. So we can define

$$[\![\text{did too}]\!] = \lambda P.\, \lambda i.\, \lambda k.\, \texttt{selVP}(i)P(i)(k)$$

In Haskell:

```
liftEllipsis :: NounPhrase -> Sentence
liftEllipsis = \np i k -> (selVP i) np i k
```

3.3 Adding Possessive Pronouns

Before we can test our interpretation functions on the example sentences, we need to add possessive pronouns to our grammar and model, as well as determine their continuized representations.

We modify our grammar by adding an additional rule for DET:

DET $\rightarrow$ *his* | *her* | *its*

Our model acquires the predicate *possess* as a simple way to model possession in our current entity-and-predicate structure:

```
possess :: TwoPlacePred
possess = \x y -> (x,y) `elem` [(X,E), (F,Y), (C,M), (H,S)]
```

This translates as the princess (`E`) possessing a (in fact, the only) dagger, and Atreyu, Little Mook and Snow White each possessing a (distinct) sword.

What is the representation of *her*? It has type `Noun -> NounPhrase`, like the determiners *every* and *some*. (As discussed, this model does not track gender, so it uses exactly the same representation for *his*, *her* and *its*.)

$$[\![her]\!] = \lambda P.\, \lambda Q.\, \lambda i.\, \lambda k.\, \iota x(P(x)(i)(k) \wedge \text{possess}(x)(\texttt{sel}(i))) \wedge P(x)(i)(\lambda i'.\, Q(x)(x :: i')(k))$$

where ιx denotes that there exists a unique object x which is possessed by the owner.

In Haskell, we implement this with a filter and the function `any`:

```
intPossPronoun :: Noun -> NounPhrase
intPossPronoun = \p q i k ->
    let owner = sel i
        match = filter (\x -> (p x) i k && (possess x owner)) entities
        value = any (\x -> (p x) i (\i' -> (q x) (addEntity x i') k)) match
    in length match == 1 && value

intDET His = intPossPronoun
intDET Her = intPossPronoun
intDET Its = intPossPronoun
```

3.4 Examples

Now we have all the pieces we need to verify that our model can handle unsaturated predicates. Working with the entities we have in the model (we have people and swords for (15), but not hands for (14) or mothers for (16)), the model now correctly predicts (15) (*Atreyu loved his sword. Little Mook did too.)* as true, and the following as false (because these facts do not hold in the model):

(17) Atreyu loved his sword. Snow White did too. [False: Snow White has a sword, but doesn't love it.]

(18) Atreyu loved his sword. Alice did too. [False: Alice does not have a sword]

(19) Alice loved her sword. Little Mook did too. [False: Alice does not have a sword]

Note that even if we make Snow White love Atreyu's sword in the model, (17) is still false, because we chose our interpretation of possessives to always be 'reflexive' i.e. always refer to the subject, rather than staying constant across the VP ellipsis.

4 VP Ellipsis Across Time

Currently, our model only handles one tense, namely past tense. We would like to expand the model to handle other times, so that we can handle VP ellipsis constructions such as

(20) Alice cheers. Little Mook does too.

(21) Little Mook will cheer. Alice will too.

(22) Dorothy cheered. Alice will too.

(23) Atreyu shuddered. Some wizard would too.

(24) Some princess laughed. Goldilocks had too/already.

(25) Some princess had laughed. Goldilocks had too.

(26) Atreyu loves his sword. Little Mook did too [until he cut himself on it.]

Some of these combinations sound marked, partly because we are limited by the verbs in our model, and laughing and cheering are normally activities rather than achievements or accomplishments. The perfect aspect works with activities, but tends to sound more marked than an achievement such as *has/had found* or an accomplishment such as *has/had built*. From the verbs in our model, *defeat* and *help* might be more suited to perfect aspect examples, but we use the one-place predicates *cheer* and *shudder* to keep the examples simple.

In (24), using *too* is a little marked: *already* sounds better. We could introduce *already* to our grammar (Too → *too* | *already*), but then we would produce other unwanted combinations such as *Alice will cheer. Little Mook will already*, unless we control which tenses/aspects *already* can appear with. As with many other issues with our grammar (see 5), we could certainly do this with features, but our toy grammar does not have tense/aspect features at present, so is too simple to handle this. Thus we will set the issue of *already* aside, and use *too* with all tenses and aspects for now. Likewise (26) carries an implicit *until* ... and sounds a little marked without it, but we will not model connectives such as *until* here.

The central observation from these examples is that the VP in the ellipsis does not carry with it the tense or aspect from the first sentence. Instead, the tense/aspect of the auxiliary in the second sentence determines the time when the second occurrence of the event happens. In other words, to model VP ellipsis with varying times, we will need to detach the tense/aspect from the VP before storing it in our context. (The syntactic ramifications of this observation are highly interesting, but because our model does not overtly deal with syntactic trees, we do not need to incorporate them into this project.)

4.1 Modifying the Grammar

We need to introduce a number of rules to our grammar to capture these productions. Here is the new grammar for VP and all its children:

VP → TV NP
VP → *laughs* | *laughed* | *cheers* | *cheered* | *shudders* | *shuddered*
TV → *loves* | *loved* | *admires* | *admired* | *helps* | *helped* | *defeats* | *defeated*
VP → AUX Inf
VP → PastAUX EN VP → AUX Too
AUX → *did* | *does* | *will* | *would*
PastAUX → *has* | *had*
INF → TINF NP
INF → *laugh* | *cheer* | *shudder*
TINF → *love* | *admire* | *help* | *defeat*
EN → TEN NP
EN → *laughed* | *cheered* | *shuddered*
TEN → *loved* | *admired* | *helped* | *defeated*

We use EN as an abbreviation for the -en forms that follow *has* and *had*, and TEN for transitive -en forms. Notice that unfortunately, for the verbs in our model, these forms look identical in English to their -ed form in the simple past. In Haskell, we will need to artificially distinguish these forms (we cannot have two data constructors for different types, `VP` and `EN`, with the same name).

This handles all tense and aspect combinations in English except future perfect: *Alice will have cheered.* Our toy grammar is too simple to handle future perfect without issues: we will discuss this in Section 5.2.

Note also that we are only concerned with the sense of *would* where it means 'in the past, it was the case that in the future X would happen' i.e. for some evaluation time t, there is a past time $t_1 < t$ such that

there is $t_2 > t_1$ such that the event in question happens at t_2. *Would* also has a conditional meaning, and of course it is possible to form VP ellipsis constructions with that too – more on this in Section 5.5.

Finally, this grammar does have the imperfection of generating sentences of the form *Alice did laugh* and *Alice does laugh*, which, without emphasis or context in dialogue, sound questionable. However these are not ungrammatical English sentences – just marked without proper context – and so we will set this issue aside in favour of keeping our toy grammar simple.

4.2 Adapting the Model

Next, we need to incorporate these tenses and aspects into our model. We will use a simple LTL model with four worlds: W1, W2, W3, W4 (so W1 is before W2 is before W3 is before W4). We will interpret simple past as the P operator, so $P\phi$ is true in a world iff ϕ is true in some world strictly before it. Likewise the perfect aspect is simply interpreted as placing the event in the past, so also with P. Of course this does not do justice to the complexity of the perfect aspect, but LTL is only concerned with the temporal order of events.

We will interpret the future as a strict future ($XF\phi$ where X moves to the next world) instead of $F\phi$, which could also be true in the current world. Likewise future perfect is interpreted as ϕ being true at any world which is in the past of some future world, and *would* (future in the past) is interpreted as ϕ being true at some world which is in the future of some past world.

Finally present tense just designates the current world (world of evaluation). Of course this also neglects the complexities of the present tense in English, notably that we normally prefer the present progressive (*Snow White is sleeping*) over the simple present (*Snow White sleeps*) if we want to express what is happening in the current world. We will not handle habitual or generic uses of the present tense here.

Since we have very specific uses of F and P, we will not implement LTL fully. The cases that we have can be simplified substantially, as shown in the Haskell code. It implements each tense/aspect as an accessibility relation from a world to a list of worlds. For example, this is the implementation of `past`:

```
data World = W1 | W2 | W3 | W4 deriving (Eq, Bounded, Enum)

worlds :: [World]
worlds = [minBound..maxBound]

type AccRel = World -> [World]

past :: AccRel
past w = init (enumFromTo minBound w)
```

That is, `past` simply returns all the worlds in our list up to, but not including the current world (note that `enumFromTo` in Haskell includes its start and end, so we need to use `init` to remove the current world. Likewise, we can simplify future perfect and past future (*would*) by realising that when we ask for all worlds in the future of some past world, or past of some future world, this returns all worlds except potentially the first and last world (the first world does not have a past world, so there are no worlds in the future of its past; similarly for the last world which does not have a future world). The interested reader is referred to the accompanying code for details.

We also need to modify our predicates to be evaluated in a given world. A one-place predicate no longer just maps an entity to a truth value: the truth value returned depends on the world it is being evaluated in.

```
type OnePlacePred = World -> Entity -> Bool
type TwoPlacePred = World -> Entity -> Entity -> Bool
```

Plausibly, we assume that common nouns and the possession relation are constant across our worlds: someone who is a girl or a dwarf or a wizard is always a girl or a dwarf or a wizard, and Atreyu's sword is always Atreyu's sword: it is his whether he is carrying it or not; he will never give it away or sell it.

However, our other predicates can change: people laugh, cheer, and defeat other people only at certain moments; they fall in and out of love, and so forth.

For example, *shudder* is now defined as follows: in W1, Snow White and Atreyu shudder, in W3, one of the wizards (`V`) shudders, and nobody else shudders at any other time.

```
shudder = \w x -> (w, x) `elem` [(W1, S), (W1, Y), (W3, V)]
```

(Again, in Haskell, this is implemented with some helper functions since we have many predicates which need to be defined similarly, and most predicates have many more occurrences in the worlds.)

4.3 Modifying our Representations

To evaluate a sentence, it needs not only a left and right context, but also a world. This world will mostly be passed down through all the representations, much like the left and right contexts, until it reaches a predicate. At that point, we evaluate the predicate not only against the relevant entities but also against the world. As the world gets passed through the representations, it will at some point undergo an accessibility relation at the point where we identify the tense and/or aspect in the sentence. So in fact, when we evaluate the predicate, we need to check whether it holds in any of the worlds returned by the accessibility relation.

In many cases, we just replace $\lambda i.\, \lambda k.$ by $\lambda i.\, \lambda k.\, \lambda w.$ For example, `compose` and `liftNP` (which implements $[\![NP]\!]$ for proper nouns) become

$$\texttt{compose}(d, s) = \lambda i.\, \lambda k.\, \lambda w.\, d(i)(\lambda i'.\, s(i')(k)(w))(w)$$
$$\texttt{liftNP}(e) = \lambda P.\, \lambda i.\, \lambda k.\, \lambda w.\, P(e)(e :: i)(k)(w)$$

The main changes lie in our representation and lifting of VPs. In `intVP`, where we map from syntactic form to our representation, we now need to pick the tense off the syntactic form and pass the appropriate accessibility relation to `liftVP` or `liftTV`. For example, `intVP Laughs = liftAndAddOnePlace laugh present` – note how the present tense has appeared as a second argument to the helper function we defined above. We also need a number of helper functions to handle mapping every combination of transitive/intransitive verb and tense/aspect, but ultimately these all call either `liftAndAddOnePlace` or `liftAndAddTwoPlace` with some predicate and accessibility relation, so we will not enumerate them all here.

As we observed, we must detach the accessibility relation from the VP before adding it to the context. Instead of adding VPs to our context, we want to add functions which take an accessibility relation and apply it to return a VP. For lack of a better name, we will call this a 'plain VP'.

```
type PlainVP = AccRel -> NounPhrase -> Sentence
data Context = Context {contextEntities :: [Entity], vps :: [PlainVP]}

selVP :: Context -> PlainVP
selVP (Context entities vps) = head vps
```

Next we modify our helper functions to lift the verb with its accessibility relation, and store it on the context without:

```
liftAndAdd :: (AccRel -> NounPhrase -> Sentence)
                -> AccRel -> NounPhrase -> Sentence
liftAndAdd boundLift acc = \np i k w ->
                            (boundLift acc) np (addVP boundLift i) k w

liftAndAddOnePlace :: OnePlacePred -> AccRel -> NounPhrase -> Sentence
liftAndAddOnePlace v acc = liftAndAdd (\t -> liftVP v t) acc

liftAndAddTwoPlace :: TwoPlacePred -> AccRel -> NP -> NounPhrase -> Sentence
liftAndAddTwoPlace tv acc objNP = liftAndAdd (\t -> (liftTV tv t)
                                    (intNP objNP)) acc
```

Finally, we can pass the world to the predicate.

$$[\![\text{cheer}, r]\!] = \lambda P.\, P(\lambda x.\, \lambda i.\, \lambda k.\, \lambda w.\, (\exists w'.r(w, w') \wedge \text{cheer}(w)(x)) \wedge k(i))$$

where r stands for the accessibility relation generated by the tense/aspect.

In Haskell, we can use `any` on the list of worlds generated by the accessibility relation instead of the constraint on $\exists w'$:

```
liftVP :: OnePlacePred -> AccRel -> NounPhrase -> Sentence
liftVP iv accRel = \np -> np (\x i k w ->
                        any (\w' -> iv w' x) (accRel w) && k i)
```

This handles tense for single sentences. Now we can turn to interpreting VP ellipsis. We were already careful to put the VP on the stack without its tense: in fact, we put a λ-expression on the stack which expects an accessibility relation and then returns a VP. When we encounter a VP ellipsis, we can take it off the stack and apply the accessibility relation given by the tense/aspect of the auxiliary.

$$[\![\text{did too}]\!] = \lambda P.\, \lambda i.\, \lambda k.\, \lambda w.\, ((\texttt{selVP}(i))(r_{\text{past}}))\ P(i)(k)(w)$$

where r_{past} is the accessibility relation for past. *Does too*, *had too* and so forth will each call the ellipsis interpretation with a different accessibility relation.

In Haskell,

```
liftEllipsis :: AccRel -> NounPhrase -> Sentence
liftEllipsis accRel = \np i k w -> ((selVP i) accRel) np i k w
```

These are all the pieces we need to interpret the varied tense VP examples from the beginning of the section:

(20) Alice cheers. Little Mook does too.

(21) Little Mook will cheer. Alice will too.

(22) Dorothy cheered. Alice will too.

(23) Atreyu shuddered. Some wizard would too.

(24) Some princess laughed. Goldilocks had too.

(25) Some princess had laughed. Goldilocks had too.

(26) Atreyu loves his sword. Little Mook did too.

A worked example is too long to include here, but the interested reader is welcome to step through the accompanying Haskell code to see how the model interprets each example.

4.4 Test Sentences and Evaluation

The Haskell code contains a wide variety of test sentences at the bottom of `VPEllipsis.hs`, including all of the examples in this report, and an exhaustive list of tense/aspect examples to ensure each one works correctly. These test sentences can be checked individually or all at once by evaluating the helper constant `isCorrect`.

5 Issues and Extensions

As hinted at throughout this report, there are some issues with this model, and many ways in which this project could be extended. The handling of context, in particular, only scratches the surface of the many aspects which influence human choice when resolving anaphora. At the heart of the project's scope for improvement is our minimalist selection function, which always takes the item at the top of the stack, regardless of other context. Before we discuss the selection function in detail, two minor comments on the grammar are in order.

5.1 Accusative Pronouns

A minor issue with our grammar is that it does not capture the difference between nominative *he, she* and accusative *him, her* – it allows *he* and *she* as NPs in all the places that NPs can occur, both in subject and object position. So if we wanted to interpret *Atreyu admires her* (see example (29) below), the grammar is only able to parse *Atreyu admired she.* By including features and thus allowing subcategorization frames in our grammar, we could restrict this – probably without impacting the semantics too heavily. It was important to first develop a model of VP ellipsis without obfuscating its mechanics through a more complex grammar, but of course to model real-world English sentences this needs to be accounted for.

5.2 Implementing Future Perfect

As mentioned when introducing tenses to the grammar, our current grammar cannot handle future perfect. If we introduce this to our grammar naïvely as

INF → AUXINF EN
AUXINF → *have*

then we get unwanted productions such as *Alice did have cheered* or *Alice does have cheered*. Again, with a richer grammar that has features and subcategorization frames, or with some slightly ugly duplication of rules, we could ensure that *have* can only follow *will* or *would* and not other auxiliaries. In this project, this would make our toy grammar much more complicated and obfuscate the point we are making about VP ellipsis. Of course, a complete account of English tenses still needs to handle this case.

5.3 Improving the Selection Function

The main scope for improvement with this model lies in the selection function. One issue with our selection function is that it handles the following cases incorrectly:

(27) Some wizard shuddered. Atreyu defeated him.

(28) Some wizard shuddered. Atreyu defeated him. Little Mook did too.

In (27), our model interprets *him* as the most recently-mentioned entity: *Atreyu*. However as English speakers we know that because the sentence uses *him* and not *himself*, it cannot refer to the agent *Atreyu* but must instead refer to some previous salient entity, in this case the previous entity *some wizard*. In (28), this is exaggerated: the first instance of *him* refers to *Atreyu* and the second to *Little Mook* when in both cases we want it to refer to *some wizard*.

In other words, our context should be keeping track of who is the agent of the current predicate and excluding it when resolving anaphora that do not have a reflexive marker (*him* vs. *himself*). It is not enough to simply track which entities are agents, as all the entities in our context are agents of some verb in this example: *some wizard* is the agent of *shuddered*. Our `sel` function for entities should check against the current VP in the context and its arguments. How would we store this information? While the current stack contains VPs, it contains a timeless copy of the current VP with only its complements and not its subject. Enriching the context to track semantic roles and/or making sure that entities do not get used more than once as arguments to a predicate (while excepting the cases where that is allowed by reflexivity or other indicators) seems both interesting and quite challenging.

As we discussed above, the selection function also doesn't handle gender features. So it selects the wrong entity in cases such as the below, where it does not realise that the first item on the stack is invalid for the pronoun and that it should skip to the next one.

(29) Snow White laughed. Little Mook did too. Atreyu admired her. [assuming Little Mook is male]

Here, our model will select *Atreyu* for *her*, because that is the most recently mentioned entity.

Both of these types of example have in common that the correct entity is the top-most *valid* entity in the stack – we "just" need to teach the model what entities are valid or invalid in a given context.

Adding gender to the model, however, potentially causes new issues. Consider

(30) Atreyu loved his sword. Alice did too.

Notice that humans have no problem changing the gender of the pronoun: we can read (30) as *Alice loved <u>her</u> sword* even though the original VP uses *<u>his</u> sword*. (However, this pronoun shift does change which interpretation we think is more likely; exactly which we pick depends on the larger context.) So does our model need to strip the gender features of possessive pronouns when storing the VP in the context?

5.4 Narrowing Down Temporal References

Another trait of our model is that it is very liberal in its interpretation of tense. While it's true that the past tense technically doesn't provide any information beyond the event occurring before the present, in fact there is often a contextually salient past time from a previous sentence in the discourse. If not, the first past tense may introduce such a time, either by its existence alone, or more likely by being accompanied by some temporal adverb. (Think of the traditional *once upon a time...*)

This becomes apparent in VP ellipsis. Consider the following example:

(30) Atreyu had laughed. Snow White would too.

In (30), our interpretation of *would* first selects a past world (any past world), then finds a future world of that where Snow White laughs. However, as famously shown by Reichenbach (1947), *had laughed* introduces a reference time which is the point before which Atreyu laughed. When we interpret *would*, we want to use the same reference time as the point after which Snow White laughed. Moreover, it is probable that, in this past-tense narrative, Snow White laughed in the future of that reference point but *before the current time*. As we have set it up, not only are the two reference times calculated independently (in fact, entirely ignored by the calculation, since they do not influence the end result), but the future time could be after the present.

To address this, we might consider adding a third part to our context, which keeps a list of salient times, in addition to salient entities and salient VPs. Each event adds its event time and, if it has one, its reference time to the list. When processing a new event which has a reference time, we should consider drawing an existing reference time or event time from the list instead of creating a new one.

In fact, there are cases where not only the reference time but the event time is fixed.

(31) Atreyu laughed. Snow White did too.

(32) Atreyu laughed at the joke. Snow White did too.

Consider (31). Did Atreyu and Snow White laugh at the same time? Given no further context, it seems likely, but we cannot be sure. So our model may be overzealous in treating them completely independently and checking all past times for both of them with `any`, but it is not wrong. However in (32), the event times should be the same. This goes a step further than the case above where the reference times needed to match. Because the VP ellipsis inherits the complement *the joke*, it gets anchored to the point in time when the joke happened – and *the* introduces a single, unique event time we can make reference to. There are many reasons why we didn't cover *the* in this model, and this is one of them: besides its traditional stipulation of uniqueness, it also introduces new entities or events into the discourse which can be referred to later, which our context would have trouble with.

Much more work is needed in this area to determine under what conditions the event time or reference time of the VP ellipsis is fixed by the context and under what conditions it is free (constrained only by the tense/aspect). Armed with that data, a very interesting project could begin to decide how to incorporate times into the context and how the selection function should behave.

5.5 Double Ellipsis Constructions in Conditionals

As noted in Section 4.1, we can model the meaning of *would* as a past future (as in *Atreyu shuddered. Some wizard would too.*) However, there is a second sense of *would* where it is used in conditionals and counterfactuals:

(33) If Atreyu went to the castle, he would meet a princess.

(34) If Atreyu had gone to the castle, he would have met a princess.

(35) If Atreyu went to the castle, he would meet a princess. If Dorothy went to the castle, she would too.

(36) If Atreyu went to the castle, he would meet a princess. If Dorothy did, she would too.

(37) If Atreyu had gone to the castle, he would have met a princess. If Dorothy had gone to the castle, she would (have) too.

(38) If Atreyu had gone to the castle, he would have met a princess. If Dorothy had, she would (have) too.

These cases are quite interesting. Firstly, note the double ellipsis in (36) and (38): how would we build into the model which ellipsis refers to which VP? Secondly, observe how in (37) and (38), we can have one or two auxiliaries before *too* – our toy grammar will need to handle that, perhaps similarly to handling future perfect, and so will our interpretation functions, which may prove more complicated. These cases would make an excellent topic to study, but are beyond the scope of this project: to handle these, we'd first need to extend our model to handle conditionals and counterfactuals – an entire project in and of itself.

References

Asher, Nicholas and Sylvain Pogodalla (2010). "SDRT and Continuation Semantics". In: *JSAI International Symposium on Artificial Intelligence*. Springer, pp. 3–15.

Barker, Chris and Chung-chieh Shan (2014). *Continuations and Natural Language*. Vol. 53. OUP Oxford.

Cann, Ronnie, Ruth Kempson, and Eleni Gregoromichelaki (2009). *Semantics: An Introduction to Meaning in Language*. Cambridge Textbooks in Linguistics. Cambridge University Press.

De Groote, Philippe (2006). "Towards a Montagovian Account of Dynamics". In: *Semantics and Linguistic Theory*. Vol. 16, pp. 1–16.

Reichenbach, Hans (1947). *Elements of Symbolic Logic*. Dover Publications, Inc.

Van Eijck, Jan and Christina Unger (2010). *Computational Semantics with Functional Programming*. Cambridge University Press.