Computational Linguistics Final Project
Quantifier Scope and Continuations

Chris Anderson

December 11, 2013

1 Introduction

In this paper, we examine the computer science idea of continuations, and how, as suggested by Chris Barker, they are a solution to many of the challenges in computational linguistics, specifically in semantics. We implement a simple context-free grammar using continuations, and use it to illustrate how one such difficult semantic problem, inverse quantifier scope, is resolved.

2 Continuations

One of the most common definitions of a continuation is that it is "the entire default future of a computation". While technically correct, this perhaps does not capture why computer science has found them useful.

In essence, a function that is written using continuations can save the context that it is being executed in, packaging all of the necessary information to complete its evaluation into a continuation, while not actually being evaluated to a final value. Thus, continuations give us a way of deferring computations, by passing that continuation to other functions and finding its value later, and in fact of being able to perform the actual evaluation in a program in any order we want.

This makes them one of the most powerful tools at a programmer's disposal for flow control, other than of course unrestricted use of gotos.

2.1 Continuation passing style

Many programming languages, most notably Scheme, have first-class continuations, where the language is able to automatically determine what our "default future" is and turn it into an object. However, OCaml does not have this feature, so we must determine the correct continuations and write them by hand, writing code in what is known as continuation passing style. This will turn out to be beneficial to our use of continuations for semantics, as the explicitly stated form of a continuation used in CPS makes it clearer why they apply to computational semantics.

The nature of continuation passing style is probably best illustrated by writing a trivial OCaml function both in the normal style and in CPS. Say we want to write the List.map function from scratch, which takes a list and a function and returns the list with the function applied to every element.

As normally written:

```
let rec map f lst = match lst with
  | [] -> []
  | x::xs -> (f x)::(map f xs)
```

Converted to CPS:

```
let rec map_cps f lst c = match lst with
  | [] -> c []
  | x::xs -> map_cps f xs (fun a -> c ((f x)::a))
```

In the CPS version, rather than explicitly computing (f x) before the next function call, we create a function, the continuation, which will evaluate to (f x) being prepended to a list, when we evaluate it. That is the default future in this context. By actually evaluating the continuation at the end, we force every one of those deferred operations to be evaluated.

Note that to ensure a CPS function returns the same value as its normally-written equivalent, it should be passed the identity function fun x → x when first invoked.

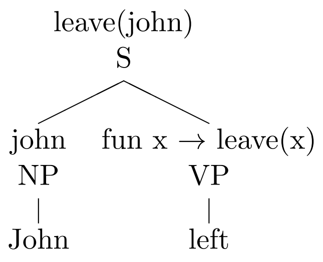
In general, given some arbitrary value of type 'a, the continuation passing style equivalent is a function which takes a function and evaluates it with that value, for type ('a → 'b) → 'b.

One of the most basic uses of continuations in linguistics is as a way of implementing a parser which avoids infinite recursion on left-recursive rules by delaying evaluation with continuations. But there are other ways to accomplish that, and it only uses them in the programming sense to simplify the task, while continuations are in fact part of linguistics on a much more fundamental level.

3 Semantics and Quantifiers

In general, when doing semantics we would like to be able to classify sentences as whether their meaning is true or false. The most basic way to handle semantics is that NPs become "entities", and VPs describe properties that entities can have. By defining a property as a function from entities to truth values - whether it is true that the entity possesses that property - we are able to easily evaluate simple sentences by applying the VP to the NP, and obtaining a truth value for the entire sentence.

For example, if we have the sentence "John left", it would be written like this:

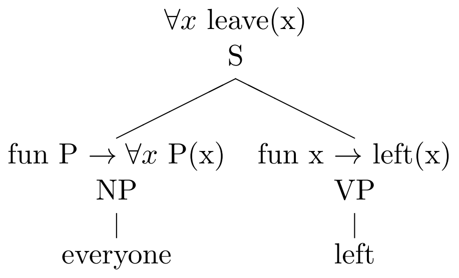


which would have a truth value according to whether or not John did in fact leave.

The primary difficulty with this arises when one considers quantifiers - statements of the form "every x" or "some x", since we cannot pass $\forall x$ or $\exists x$ as an entity to the property function.

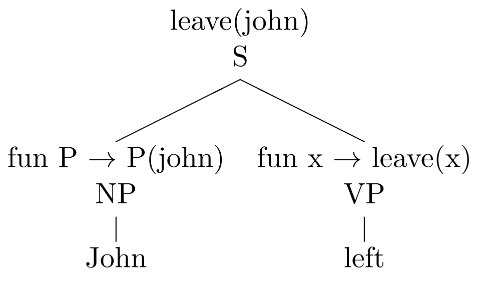
Richard Montague proposed a solution to this in which the relationship is reversed: instead of properties taking entities as an argument to return a truth value, entities are functions that take properties as their argument. Thus, "everyone" is understood to be a function which, given a property, checks the truth value of that property for all entities, and returns whether it is true for all of them.

Then, we can express a simple sentence that uses a quantifier by:



The VP is given to the NP, which sets what property P is, and then that property is checked for all entities to determine the truth of the sentence.

Where Barker noticed the idea of continuations specifically present in linguistics, though, is the novel way that Montague decided to handle proper nouns. He simply raises NPs that aren't quantified to the same type as ones that are quantified. So "John" is a function that takes a property and returns whether it is true for John.



But this is effectively equivalent to writing "John" in continuation passing style.

Recall that a continuized value is of type ('a → 'b) → 'b. If we say that John is of type entity, then a property like "left" is of type (entity → bool).

So that makes the type of a NP in Montague's grammar: (entity → bool) → bool. And as Barker says, that matches continuation passing style exactly, despite Montague never referring to it as a continuation; he remarks that continuations were invented by multiple people independently around the same time, and that Montague was yet another researcher to hit on the idea.

Looked at from the point of view of how we define continuations in computer science, NPs having the continuized type (entity → bool) → bool makes perfect sense. The context that a NP is evaluated in is a sentence that has to have a VP; the default future from having seen a NP is that the VP is going to happen to it.

4 Continuizing a grammar

According to Barker, Montague's idea is really just a generalized case of what he calls a continuized grammar: a grammar in which all terms have been transformed into continuation passing style, not just the NPs. In this section, we implement such a grammar in OCaml.

Here is the grammar in its original form, adapted from the examples Barker uses in his paper:

```
S → NP VP
NP → Det N
VP → Vt NP
NP → John
NP → Mary
(...some more names...)
NP → everyone
NP → someone
Det → every
Det → a
N → man
N → woman
VP → left
Vt → saw
```

If we start with names denoting entities and other terms being functions from entities to truth values, then assuming we define an entity type in the code (which is simply type entity = John | Mary | ...) we get these as our data types that we should have before and after continuizing:

Category	Type	Continuized Type
S	bool	(bool → bool) → bool
NP	entity	(entity → bool) → bool
VP	e → b	((e → b) → b) → b
Vt	e → e → b	((e → e → b) → b) → b
Det	(e → b) → (e → b) → b	(((e → b) → b) → b) → (e → b) → b
N	e → b	((e → b) → b) → b

We have already reasoned through why NP has those types. As for the others...

A continuized S is simply the truth value of the sentence, packaged in a continuation so that we have to apply the identity function to get it.

A continuized VP, being a property, necessarily takes a continuized NP as its argument, and evaluates it to a truth value.

Transitive verbs require an entity first as their object, after which point they are identical to VPs.

Nouns like "man" or "woman" when used in the Det N context of "every man" etc, are really just a property in the same way that VPs are a property.

Determiners take a noun and are then equivalent to a NP.

4.1 Implementation

How are we to go about actually writing this grammar in OCaml? A logical first step is to write the non-quantificational parts of the grammar in the naive way we would write them if we didn't have to worry about quantifiers, then convert them to continuation passing style. For anything that is possible to express without Montague's method for handling quantifiers, we can just automatically convert it to CPS:

```
let cps_terminal x cont = cont x
```

Then, assuming we have defined our entities (type entity = John | Mary | ...), and defined functions over entities for our verbs (let leave = function John → true | ...), using cps_terminal will properly convert any single term that isn't quantified.

```
let john = (NP, (cps_terminal John))
let left = (VP, (cps_terminal leave))
```

will work exactly as they should. Note that I added a category in front of all terms so that binary rules like S → NP VP could determine if they were being passed something that makes sense in the grammar.

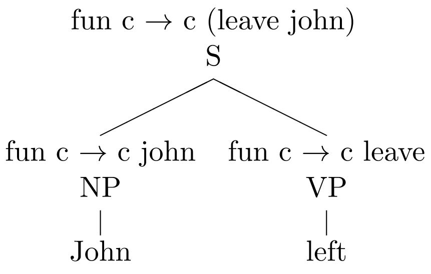
To evaluate those binary rules, we need the ability to perform function application when both the function and its argument are in CPS form. I needed van Eijck and Unger's section on continuation passing to understand how this works because it looks unintuitive, but the basic idea is that we evaluate one and continuize the result, then use that continuation with the other.

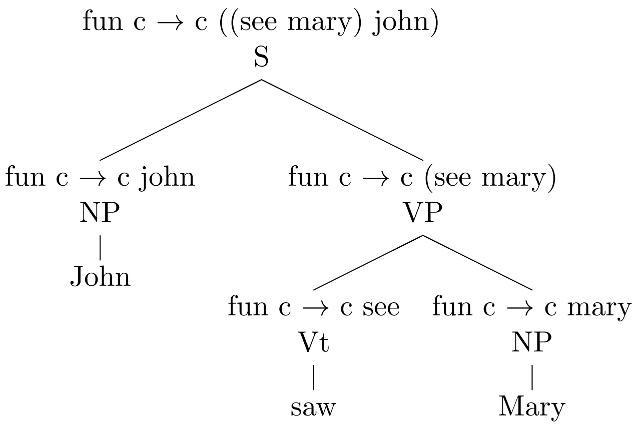
```
let cps_apply f g = fun cont -> g (fun b -> f (fun a -> cont (a b)))
```

That makes binary rules a matter of checking whether the categories are correct, then cps_applying the two continuations, for example:

```
let sentence np vp = match (np, vp) with
  | ((NP, a), (VP, b)) -> (S, (cps_apply b a))
```

With all the binary rules so continuized, we can check the truth values of sentences that don't have quantifiers using the continuized grammar.

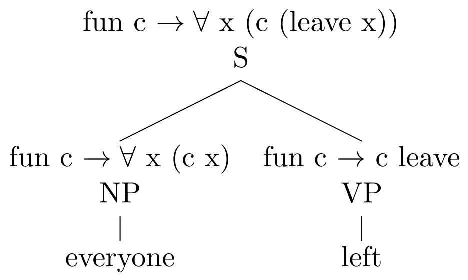


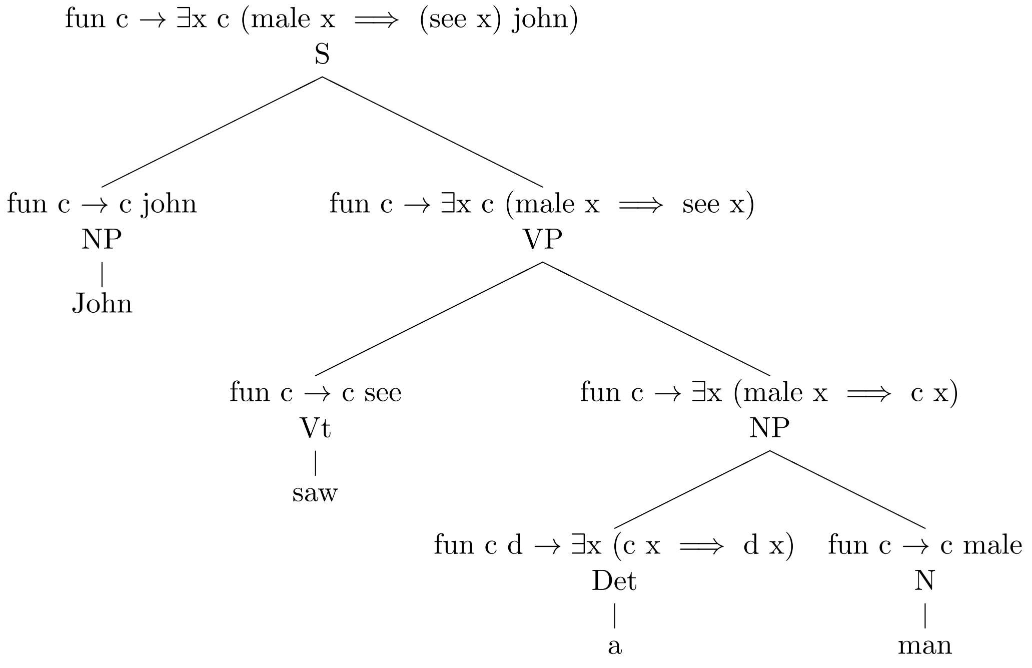


Both of these, when the result is applied to the identity function, will yield the correct truth value for the sentence.

4.2 Quantifiers

Quantified NPs fit into the continuized model nicely, because the type of applying a continuation to an entity to get a truth value is the same as applying a continuation to a list of entities, and checking if any/all of them are true. Assuming we have a list of all valid entities, then the built-in OCaml functions List.for_all and List.exists get the job done of allowing us to write fun c → ∀ x (c x) and fun c → ∃ x (c x). For "every x" in a NP → Det N, we use the property given by the N to filter the list of entities down to only those that have the property, then proceed as above.




fun c → ∃x c (male x ⟹ (see x) john)
S
fun c → c john
NP
John
fun c → ∃x c (male x ⟹ see x)
VP
fun c → c see
Vt
saw
fun c → ∃x (male x ⟹ c x)
NP
fun c d → ∃x (c x ⟹ d x)
Det
a
fun c → c male
N
man

4.3 Inverse scope

Finally, we show that, as stated in the introduction, the problem of ambiguity known as inverse scope is solved very simply using continuations, something that Barker noted was one of the benefits of continuizing the entire grammar rather than just the NPs as Montague did.

The problem, essentially, is that when quantified NPs are present in both the NP and the VP (the latter occurring as the object of a transitive verb), the sentence has a different meaning depending on which quantifier we give precedence over the other, if one is $\exists$ and the other is $\forall$. Normally one assumes that a sentence uses linear scope - the quantifier in the subject, which we see first, takes precedence, but it is equally valid for the sentence to have inverse scope, where the quantifier in the object takes precedence.

To illustrate this, in the grammar we have been using assume that we have four people who are valid entities, John, Dave, Mary, and Anna, and assume that John saw Mary but not Anna and Dave saw Anna but not Mary. Then the meaning of "every man saw a woman" depends on scope.

Linear scope	(every man (saw a woman))	$\forall x(male(x) \wedge \exists y(female(y) \wedge saw(y,x)))$
Inverse scope	((every man) saw a woman)	$\exists x(female(x) \wedge \forall y(male(y) \wedge saw(y,x)))$

The first means "every man saw some woman, possibly different than who the other men saw", and is true given the truth values we assigned above, whereas the second means "there is a woman who was seen by all of the men", which evaluates to false.

As we know, one of the benefits of continuations is that they allow us to evaluate expressions when we want and in the order that we want. In particular, when implementing cps_apply, we implicitly made a choice about what order we wanted expressions to be evaluated in.

```
let cps_apply f g = fun cont -> g (fun b -> f (fun a -> cont (a b)))
```

was our function to apply f to g when both are written in CPS style. Since the rule for S → NP VP is evaluated in CPS by applying VP NP, the VP becomes the inner f and the NP becomes the outer G in cps_apply. That automatically gives the NP scope over the VP, which gives us linear scope.

But the order of the terms in that function was essentially arbitrary. If we swap them, it still works.

```
let cps_apply' f g = fun cont -> f (fun a -> g (fun b -> cont (a b)))
```

When the sentence rule is modified to use this instead, the VP is outermost and gets scope over the NP, causing inverse scope.

The results of setting the assumed truth values of who saw who and checking both interpretations (sentence' is the version of the rule that uses cps_apply') show that this works.

```
# eval (sentence (np every man) (vp saw (np a woman))) identity;;
- : bool = true
# eval (sentence' (np every man) (vp saw (np a woman))) identity;;
- : bool = false
```

Therefore, simply by using both orders of writing continuized application to get both orders of evaluation, we can evaluate the two distinct meanings of a sentence whose quantifiers can be either linearly or inverse scoped, without having to deal with the ambiguity normally inherent in the problem. By introducing the concept of a continuation, we have turned inverse scope from a problematic ambiguity that one accepts, to a natural feature of the way we evaluate things in continuation passing style that we can handle with relative ease.

5 Further work and conclusions

Although I was unable to get a continuized grammar for them working, Barker suggests in another paper that continuations could be used to evaluate the sentences known as "donkey sentences". These are sentences with the structure of "every farmer who owns a donkey beats it", which do not intuitively lend themselves to being converted to logic in the normal manner. While "a donkey" would normally indicate some arbitrary one and would by most methods end up as $\exists$, the VP indicates that it must mean the specific donkey owned by the farmer. Thus, these sentences are properly interpreted as "if a farmer owns a donkey, he beats that same donkey" or "there is no farmer who owns a donkey and does not beat the donkey", statements where either both quantifiers are $\forall$ or both $\exists$.

Barker outlines an extension of his idea of continuized grammars, which among its new features includes the ability to indicate that a NP and pronoun in the sentence are referring to the same entity (needed for the "beats it" structure), and uses this to evaluate donkey sentences.

Overall, while I would have liked to have been able to implement the necessary modifications for donkey sentences, and given more time I would have done so, I feel that implementing continuations for inverse scope was both a sufficiently challenging functional programming task, and an interesting exploration of the mechanics behind a form of ambiguity introduced in lecture but not fully explained in depth as to why it works.

6 Bibliography

Chris Barker, Chung-chieh Shan. "Continuations." http://esslli2004.loria.fr/content/readers/24.pdf

Chris Barker, "Continuations in Natural Language". http://www.cs.bham.ac.uk/ hx-t/cw04/barker.pdf

Chris Barker, "Donkey anaphora is in-scope binding" http://semprag.org/article/view/sp.1.1/74

Jan van Eijck and Christina Unger, Computational Semantics with Functional Programming, Cambridge University Press, 2010.

Section 4.4.4 of course notes "Charts and encoding flow of control (optional)"
https://courses.cit.cornell.edu/ling4424/notes/hale-notes004.html#toc20

CS 3110 course notes on continuation passing style
http://www.cs.cornell.edu/courses/cs3110/2013sp/recitations/rec25-cps/rec25.html