CHAPTER 12

Computational connections

In this chapter we address two of the more overtly computational aspects of our project. First, we will consider order of evaluation from the point of view of the pure lambda calculus, one of the original inspirations for our approach.

Second, we will provide some notes on a computational implementation of the tower system, including a refactoring of the type-shifters into a group of operators that provide a effective way to search for analyses.

12.1. Order of evaluation in the lambda calculus

In computer programming languages, order matters.

(203) a.
```
x := x + 1
x := 1
print x
```
b.
```
x := 1
x := x + 1
print x
```

These two blocks of pseudocode differ only in the order of the first two statements. Nevertheless, the behavior of the two blocks will differ: the first will print the number 1, and the second will print the number 2.

What we need in order to understand this difference is some way of representing the meaning of these blocks that encodes the difference in evaluation order. The danger is that the language that we use to encode the difference will itself be order-sensitive, in which case our analysis is in danger of being circular: how can we make sure that the encoding language will be evaluated in the right way?

In order to make this problem concrete, consider representing code in the pure lambda calculus. The lambda calculus is confluent, which means that no matter which application you decided to reduce first, you can always eventually arrive at the same result. Could the lambda calculus, then, serve as an order-neutral representation of sequential computations?

The problem is that the pure lambda calculus is not completely order-neutral. If a lambda form contains subterms that do not have a normal form, the order in which we choose to reduce applications matters very much, as we will see shortly.

First, we need to be more explicit about when we will consider a lambda term to have been fully evaluated. Following Plotkin (1975), we will classify the

constructions of the lambda calculus into values and programs:

	x	Variable: value
(204)	$(\lambda x M)$	Abstract: value
	(MN)	Application: program

Here, x schematizes over variables, and M and N schematize over arbitrary lambda terms. We will consider our job of evaluating an expression to be complete if the reduced term is a value, that is, either a variable or a lambda abstract. But if the term is an application, we must reduce it if possible, continuing until we have a value.

Some terms cannot be reduced to a value:

$$\Omega = ((\lambda x(xx))(\lambda x(xx))) \tag{205}$$

Because this term is an application (i.e., is of the form (MN) where $M = N = \lambda x.xx$), it is a program, and not fully reduced. If we attempt to beta-reduce this form, we get a "reduced" form that is identical to the original form (up to alphabetic variance). The result is itself an application, and therefore not fully reduced. This is the simplest "infinite loop" in the pure lambda calculus.

Now we can construct a lambda term where order of reduction makes a big difference. Let $\mathbf{I}$ be the identity function (λxx). Then

$$(\lambda x\mathbf{I})\Omega = (\lambda x(\lambda xx))\ ((\lambda x(xx))\ (\lambda x(xx))) \tag{206}$$

This is an application, so we need to reduce. However, we have a term with the form $(M(NN))$, and so have two choices for what to reduce: we can reduce the leftmost (also, outermost) application first, resulting in (λxx), a value. Success! Or we could start by reducing the rightmost (innermost) application, in which case we might begin an endless series of profitless reductions, never getting closer to a value.

Obviously, in this case, we want to start on the left. As long as we agree to always reduce the leftmost application first, we can be sure that we will always arrive at a value (that is, for any expression that reduces to a value).

But, just as in our discussion of crossover in natural language, we'd like to make this policy of always working from left to right explicit and precise. Perhaps we'd like to make it precise by encoding it in the form of a program. And since the pure lambda calculus is Turing complete, we might naturally choose to encode the reduction algorithm in the pure lambda calculus. But if we do this, then the reduction algorithm will be expressed in a language whose semantic behavior, as we have just seen, depends on evaluation order, and we're right back where we started.

What we are wishing for is an evaluation strategy for which the order of evaluation of the analyzed expression is independent of the evaluation order of language in which the analysis is conducted. This is where continuations come in.

We adopt the following strategy of Plotkin (1975): for any given term in the language to be analyzed, we map it to a different term via some explicit mapping strategy, then we reduce the expanded term. The maps that Plotkin studied are known as Continuation-Passing Style (CPS) transforms, and they make explicit use of continuations.

It turns out that a carefully-crafted CPS transform can faithfully simulate the same series of reductions that the original term would undergo given a leftmost reduction scheme (or rightmost reduction, or some more complicated regime). Yet the transformed term itself will be completely insensitive to the reduction strategy under which it is evaluated. The way the transform accomplishes this is by creating a term in which there is always exactly one redex, i.e., exactly one choice for which application to reduce next. It doesn't matter whether we reduce the leftmost redex, or the rightmost, since there is only one option. How this works will become clear as we develop the discussion in more detail.

Here is the call-by-name (CBN) continuation passing style (CPS) transform from Plotkin (1975):153. If ϕ is some term in the pure lambda calculus, we'll write $[\phi]$ for the call-by-name CPS transform of ϕ.

$$\begin{aligned} [x] &= x \\ [\lambda x M] &= \lambda\kappa.\kappa(\lambda x[M]) \\ [MN] &= \lambda\kappa.[M](\lambda m.m[N]\kappa) \end{aligned} \tag{207}$$

Thus $[\cdot]$ maps lambda terms into (more complex) lambda terms. For instance, the mapping rules apply to our problematic term as follows:

$$\begin{aligned} [(\lambda x\mathbf{I})\Omega] &= \lambda\kappa.[\lambda x\mathbf{I}](\lambda m.m[\Omega]\kappa) \\ &= \lambda\kappa.(\lambda\kappa.\kappa(\lambda x[\mathbf{I}]))(\lambda m.m[\Omega]\kappa) \end{aligned} \tag{208}$$

Crucially, this transformed term encodes the computation expressed by the original term in such a way that the the desired order of evaluation is explicitly encoded. Since the transform is an abstract (it begins with "$\lambda\kappa$"), it's already a value, so we can't evaluate the transformed expression directly. In order to see the encoding unfold, we must apply this term to the trivial continuation, $\mathbf{I}$, which triggers the reduction process. As reduction proceeds, unlike in the original term, at each stage there is exactly one possible reduction, so we can't possibly make the wrong choice:

$$
\begin{aligned}
([(\lambda x\mathbf{I})\Omega]\mathbf{I}) &= (\underline{(\lambda\kappa.(\lambda\kappa.\kappa(\lambda x[\mathbf{I}]))(\lambda m.m[\Omega]\kappa))}\mathbf{I}) \\
&= (\underline{(\lambda\kappa.\kappa(\lambda x[\mathbf{I}]))}(\lambda m.m[\Omega]\mathbf{I})) \\
&= (\underline{(\lambda m.m[\Omega]\mathbf{I})}(\lambda x[\mathbf{I}])) \\
&= (((\lambda x[\mathbf{I}])[\Omega])\mathbf{I}) \\
&= (((\lambda x[\lambda xx])[\Omega])\mathbf{I}) \\
&= (((\lambda x(\lambda\kappa.\kappa(\lambda x[x])))[\Omega])\mathbf{I}) \\
&= (((\underline{\lambda x(\lambda\kappa.\kappa(\lambda xx))})[\Omega])\mathbf{I}) \\
&= (\underline{(\lambda\kappa.\kappa(\lambda xx))}\mathbf{I}) \\
&= (\underline{\mathbf{I}}(\lambda xx)) \\
&= (\lambda xx)
\end{aligned}
\tag{209}
$$

Some of these steps are beta reductions, and some are further unfolding of the CPS transform '$[\cdot]$'. For each reduction, we have underlined the only functor that is ready to be applied to an argument. Note that the underlined lambda term is always the leftmost functor. As the zipper of evaluation descends, there is at most one application that is not hidden underneath an abstract (recall that since abstracts are already values, we don't perform reductions inside of an abstract). As a result, at each step there is only one possible move. The CBN CPS transform forces us to always reduce the leftmost application first, ignoring the inner structure of the argument Ω.

Incidentally, the reason it makes sense to call this transform "call-by-name" is that the argument (in this case, the argument is Ω) is passed to the functor unevaluated. In linguistic terms, it corresponds to a de dicto interpretation: if Lars wants to marry a Norwegian, then the description "a Norwegian" is incorporated into the description of his desire unevaluated, rather than first picking out a specific de re individual (i.e., by evaluating the direct object, call-by-value) and then supplying the individual to build the desired representation. (This analogy to the de dicto/de re ambiguity is meant to spur intuition, and should not be taken too seriously in its simplest form.)

Exercise 30: Here is Plotkin's call-by-value transform:

$$[x] = \lambda\kappa.\kappa x$$
$$[\lambda xM] = \lambda\kappa.\kappa(\lambda x[M])$$
$$[MN] = \lambda\kappa.[M](\lambda m.[N](\lambda n.mn\kappa))$$

Show that this CPS transform forces evaluation of the argument first, guaranteeing the evaluation of (206) will result in an infinite loop.

In order to see the family resemblance between Plotkin's transforms and the continuation-based system here, consider the semantic part of the combination schema from (16):

$$\frac{g[\,]}{f} \cdot \frac{h[\,]}{x} \rightarrow \frac{g[h[\,]]}{f(x)} \tag{210}$$

Translated into flat notation, this is

$$\lambda\kappa.g[\kappa f] \cdot \lambda\kappa.h[\kappa x] \Rightarrow \lambda\kappa.g[h[\kappa(fx)]] \tag{211}$$

The combinator that will produce this result given the left hand value and the right hand value is $\mathbf{C} = \lambda MN\kappa.M(\lambda m.N(\lambda n.\kappa(mn)))$. To see this, compare the following to (210).

$$\begin{aligned} \mathbf{C}MN &= (\lambda MN\kappa.M(\lambda m.N(\lambda n.\kappa(mn))))(\lambda\kappa.g[\kappa f])(\lambda\kappa.h[\kappa x]) \\ &= \lambda\kappa.g[h[\kappa(fx)]] \end{aligned} \tag{212}$$

The final result is exactly the linear presentation of the semantics of the result expression of the combination schema.

The details of **C** are not exactly the same as either Plotkin's call-by-name transform or his call-by-value transform. For one thing, Plotkin's transform makes the continuation variable κ an argument ('$mn\kappa$' in both transforms), but κ is in functor position for us ('$\kappa(mn)$'). Subtle differences in CPS transforms make for significant differences in behavior; nevertheless, the resemblance should be clear.

The main point here is that, just as a CPS transform allows explicit control over order of evaluation in the lambda calculus, so too does articulating an ordinary categorial grammar into continuation layers allow control over order of evaluation in our natural language fragment. And, just as Plotkin was able to reason about evaluation order by transforming an order-sensitive term into an order-insensitive version, so too are we able to reason about the processing order of composition by transforming the natural language expression in question into an expression in a semantic representation that is itself order-neutral.

The result is a theory in which a certain aspect of processing is represented in terms of a competence grammar. In this sense, then, we treat crossover and

other order effects simultaneously as processing defaults and as part of the competence grammar that composes well-formed expressions. On this view, there is no contradiction between being a matter of processing, and simultaneously being a matter of linguistic competence, i.e., a matter of grammaticality.

12.2. Notes on a computational implementation

The tower system was designed to make it easy to construct derivations by hand. Nevertheless, it is still useful to have a machine supply most of the details of a derivation automatically. This enables quick checking of the soundness of an analysis, and also helps find all alternative parses and interpretations for an ambiguous string.

We will show how the pressures of building a practical implementation can lead to an equivalent but different set of type-shifters. The differences will help reveal how the combinators (lifting, lowering, combination, etc.) interact with one another.

The main engineering pressure comes from the fact that the search space for derivations is not bounded. For instance, since the input pattern for the LIFT type-shifter is alway met, there is no limit to the number of times LIFT can be applied, so there is no limit to the number of distinct derivations for any sentence generated by a tower grammar. The challenge, then, is to figure out how to LIFT when needed, without LIFTing ad infinitum.

Our strategy here is to compile the LIFTing operation into the combination schema. The idea is that rather than LIFTing spontaneously, we will only LIFT when motivated by the need to combine with a neighboring tower.

Given that there are multiple options for LIFTing, the possible ways of combining any two expressions is a relation, not a function. Then let '$\Rightarrow$' be a three place relation between an expression in category A, an adjacent expression in category B, and the combined expression C (written '$A \cdot B \Rightarrow C$', pronounced 'an A merged with a B forms a C'). Then we can replace the type-shifters given above with the following combination rules: