In this paper we will sketch how an approach to semantics that in many of its aspects derives from the one pioneered by Montague (1974) can be used to assign vector meanings to linguistic phrases. The theory will be based on the simply typed lambda calculus and as a result will be neutral with respect to the linguist’s choice of syntax, in the sense that it can be combined with any existing syntax-semantics interface that assumes that the semantics is based on lambdas (e.g. linguistic trees + ‘shake ‘n bake’, f-structure + ‘glue’ in LFG, proofs + ‘semantic recipes’ in Lambek Categorial Grammar, etc.).

It will be possible to develop a stand-alone compositional vector semantics along the lines sketched here, but from the perspective of the underlying theory it will also be quite natural to have such a vector semantics work in tandem with a truth-conditional or dynamic one and let the two modules model different aspects of meaning. Distributional semantics is particularly apt at modeling associative aspects of meaning, while truth-conditional and dynamic forms of semantics are good at modeling the relation of language to reality and at modeling entailment. It is quite conceivable that a theory that combines the two as separate modules will be simpler than one that tries to make one of the approaches do things it was never intended for.

We are after a compositional vector semantics, but this paper is theoretical. So what we will not do is settle upon—from the armchair so to speak—a set of concrete vector composition operations for combining phrases or concrete matrices or cubes that embody such functions. The jury is still very much out on what are the best operations here. Mitchell and Lapata (2010) consider pointwise addition and multiplication of vectors, matrix multiplication is used in Baroni and Zamparelli (2010), while Coecke et al. (2010) and Grefenstette and Sadrzadeh (2015) employ tensor contraction. Such operations will be available to our theory, but we will not choose between them. Instead of this we will explore the question how to combine such functions once an initial set of them has been established (and validated empirically). Functions in the initial set will typically combine vector meanings of adjacent phrases. Our aim, like the one in Baroni et al. (2014) (who also give an excellent introduction to and survey of the work that has been done in compositional vector semantics), is to give a general theory that also includes dependencies between phrases that are not adjacent, such as in topicalisation and relative clause formation.

We will base ourselves on the Lambda Grammars of Muskens (2001, 2003), which were independently introduced as Abstract Categorial Grammars (ACGs) in de Groote (2001). An ACG generates two languages, an abstract language and an object language. The abstract language will simply consist of all linear lambda terms (each lambda binder binds exactly one variable) over a given vocabulary typed with abstract types. The object language has its own vocabulary and its own types. It results from 1) specifying a type homomorphism from abstract types to object types and 2) specifying a term homomorphism from abstract terms to object terms. The term homomorphism must respect the type homomorphism. A minor difference between Lambda Grammars and ACGs is that the former do not impose the requirement that object terms must (also) be linear and unless the formalism is used for syntactic description (which is not relevant here) the linearity of abstract terms can also be dropped. For more information about the procedure of obtaining an object language from an abstract language, see the papers mentioned or the explanation in Muskens (2010).

In order to provide an interpretation of our object language the type theory used must be able to talk about vectors over some field, for which we choose the reals. We need a basic object type \( R \) such that, in all interpretations under consideration, the domain \( D_R \) of type \( R \) is equal to or ‘close enough’ to the set of reals \( \mathbb{R} \) and such that constants such as \( 0 : R \), \( 1 : R \), \( + : RRR \), \( \cdot : RRR \), and \( < : RRt \) (we drop the \( \to \) in types and association is to the right; constants such as \( +, \cdot, \) and \( < \) will be written
between their arguments) have their usual interpretation. This can be done by imposing one of the sets of second-order axioms in Tarski (1946). Given these axioms $D_R = \mathbb{R}$ in full models, while we get non-standard models under the Henkin interpretation.

Vectors can now be introduced as objects of type $\mathbb{IR}$, where $I$ is interpreted as some finite index set. Think of $I$ as a set of words; if a phrase is associated with a vector $v : \mathbb{IR}$, $v$ assigns a real to each word, which gives information about the company the phrase keeps. We abstract from the order present in vectors here. Since $\mathbb{IR}$ will be used often, we will abbreviate it as $V$. Note that $\mathbb{IR}$, abbreviated as $M$, can be associated with the type of matrices and $\mathbb{IIR}$, abbreviated as $C$, with the type of cubes, and in general $I^nR$, abbreviated as $T^n$, with the type of a tensor of rank $n \geq 4$. (In this short exposition we will work with a single index type, but in the paper several index types will be considered, so that phrases of distinct categories can live in their own space.)

We need a toolkit of functions combining vectors, matrices, cubes, etc. Here are some definitions. ($r$ is of type $R$; $v$ and $u$ are of type $V$; $i$, $j$, and $k$ are of type $I$; and $m$ and $c$ are of types $M$ and $C$ respectively. Indices are written as subscripts—$v_i$ is syntactic sugar for $v_i$.)

\[
\begin{align*}
\ast & := \lambda rv.i.r \cdot v_i : RVV \\
\boxplus & := \lambda vui.v_i + u_i : VVV \\
\odot & := \lambda vui.v_i : VV \\
x_1 & := \lambda mvi. \sum_j m_{ij} \cdot v_j : MVV \\
x_2 & := \lambda cvij. \sum_k m_{ijk} \cdot v_k : CVM \\
\langle \cdot | \cdot \rangle & := \lambda vu. \sum_i v_i + u_i : VVR
\end{align*}
\]

The reader will recognise $\ast$ as scalar product, $\boxplus$ as pointwise addition, $\odot$ as pointwise multiplication, $x_1$ and $x_2$ as matrix-vector and cube-vector multiplication, and $\langle \cdot | \cdot \rangle$ as inner product. In the paper we will consider further operations, such as a rotation $\rho : VVV$ that takes two vectors and produces the result of rotating their average towards the first vector. We think of this as providing a good candidate for modelling head-argument combinations.

Let us stipulate that our basic abstract types are $D$ (for determiner phrases), $S$ (for sentences), and $N$ (for nominal phrases). A very simple type homomorphism $h$ can be defined by letting $h(D) = h(S) = h(N) = V$ (there are other possibilities and we will sketch a dynamic one in which a running context can be used to disambiguate meaning). In the table below we have provided some abstract constants (first column) with an abstract type (second column) and $h$ will now assign these an abstract type as in the fourth column.

<table>
<thead>
<tr>
<th>$c$</th>
<th>$\tau$</th>
<th>$H(c)$</th>
<th>$h(\tau)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>woman</td>
<td>$N$</td>
<td>$\lambda w.(\text{woman} \times_1 v)$</td>
<td>$VV$</td>
</tr>
<tr>
<td>tall</td>
<td>$NN$</td>
<td>$\lambda v.(\text{tall} \times_1 v)$</td>
<td>$VV$</td>
</tr>
<tr>
<td>smokes</td>
<td>$DS$</td>
<td>$\lambda w.(\text{smoke} \times_1 v)$</td>
<td>$VVV$</td>
</tr>
<tr>
<td>loves</td>
<td>$DDS$</td>
<td>$\lambda wv.((\text{love} \times_2 u) \times_1 v)$</td>
<td>$VVV$</td>
</tr>
<tr>
<td>knows</td>
<td>$SDS$</td>
<td>$\lambda wv.((\text{know} \times_2 u) \times_1 v)$</td>
<td>$VVV$</td>
</tr>
<tr>
<td>every</td>
<td>$N(DS)S$</td>
<td>$\lambda vZ.Z(\text{every} \times_1 v)$</td>
<td>$V(VV)V$</td>
</tr>
<tr>
<td>a</td>
<td>$N(DS)S$</td>
<td>$\lambda vZ.Z(\text{a} \times_1 v)$</td>
<td>$V(VV)V$</td>
</tr>
</tbody>
</table>

An (example) term homomorphism $H$ will be defined by letting its value for any abstract constant in the first column be the corresponding object term in the third column (here $\text{woman}$ is of type $V$, $\text{tall}$, $\text{smoke}$, $\text{every}$, and $\text{a}$ are of type $M$, $\text{love}$ and $\text{know}$ are of type $C$, while $Z$ is a variable of type $VV$). We automatically obtain homomorphic images of any lambda term over the constants in question. E.g. $(\text{a woman})\lambda w\xi((\text{every man})(\text{loves} \xi))$ is sent to a term that is $\beta\eta$ equivalent with

\[(\text{love} \times_2 (\text{a} \times_1 \text{woman})) \times_1 (\text{every} \times_1 \text{man}) .\]

A second pair of term and type homomorphisms can send the same abstract term to, say, $\lambda w\exists y(\text{woman} yw \land x (\text{man} xw \rightarrow \text{love} yxw))$ (there is a—specific—woman that is loved by every man).

Abstract terms that can be used as translations of linguistic expressions have a lot in common with the Logical Forms of these expressions and the lambda binders in them give easy ways for treating long distance dependencies. The use of lambda terms also makes standard ways of dealing with coordination accessible to distributional semantics. The paper will provide extensive discussion of this and will give examples where the direct use of lambdas gives an edge over the approach in Baroni et al. (2014), which is based on the directed types of Combinatory Categorial Grammar.
References


