CS 135: Quantified Noun Phrases

James Pustejovsky

September 19, 2014

1 Syntax for Typed λ-Calculus

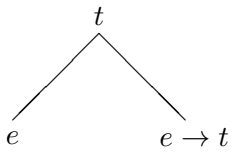
We will now present a language which uses specific types of entities, where the entities are combined with the λ-calculus. From two basic types, where e is the type of individuals and t is the type of propositions, the set T of types is generated as follows:

(1) (a) $e \subseteq T$

(b) $t \subseteq T$

(c) If $\sigma, \tau \in T$, then $\sigma \rightarrow \tau \in T$.

In order to demonstrate the application and abstraction of typed expressions, let us consider some examples from linguistic expressions. Given the basic types of e and t, we can construct recursively other types, which in fact correspond quite nicely to notions that we are familiar with from first order logic or semantics. Consider the following. If an expression such as "John", denoting the individual John in the world, is an entity and hence of type e, and an expression such as "John fell" is a proposition that is either true or false (and hence of type t), then what is the predicate "fell" in this expression typed as? Well, if the mode of construction is functional, then, it will be a function, typed as $e \rightarrow t$.



The types here are acting functionally, according to the rule of application, given below:

(2) Function Application: If α is of type a, and β is of type $a \rightarrow b$, then $\beta(\alpha)$ is of type b.

Hence, the type for an intransitive verb is $e \rightarrow t$. In fact, this is the type for any predicate in general.

(3) (i) walk: $e \rightarrow t$

(ii) walk slowly: $e \rightarrow t$

(iii) eat fish: $e \rightarrow t$

(iv) tell Mary where to buy her house: $e \rightarrow t$

The rule for abstraction allows us to substitute a predicate constant for a λ-expression of the appropriate type:

(4) (a) $walk = \lambda x[walk(x)]$

(b) $love = \lambda x \lambda y[love(y, x)]$

Notice that a binary relation is yet another function, built out of the simpler function of an intransitive predicate.

1.1 Determining the Type of an Expression

For any expression ϕ, of type τ, the functional abstraction of ϕ with variable x of type σ has the type $\sigma \rightarrow \tau$. This is visualized below:

$$\overbrace{\lambda \overbrace{x}^{\sigma} \quad \overbrace{\phi}^{\tau}}^{\sigma \rightarrow \tau}$$

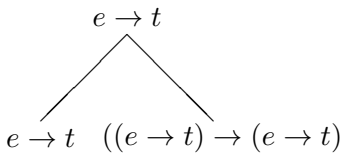
So, it might be clear how to make unary, binary, and ternary relations using functional abstraction and types, but what do we do for cases that appear to be higher order in nature? For example, consider the behavior of adverbs.

(5) (a) John fell slowly.

(b) Mary talked quickly.

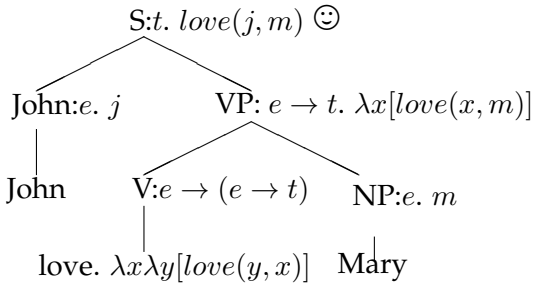
(c) Bill ate noisily.

What we will do is treat adverbial phrases as functions over predicates, returning predicates. As such, the type for an adverbial is $(e \rightarrow t) \rightarrow (e \rightarrow t))$. The type tree for such a modification is shown below:



This will be the general strategy in most typed λ-calculi for treating higher order modification relations. An example of the syntactic structure and type tree for a simple binary relation taking proper names is illustrated below.

(a) John loves Mary.



2 Semantics for Typed λ-Calculus

We define the domain of interpretation for a domain D to be $\mathcal{D}_D$, and for a type σ, the domain of interpretation is $\mathcal{D}_\sigma$. Together, we get the domain of interpretation for a type σ for a domain D, written as $\mathcal{D}_{D,\sigma}$. Since we have three type statements, there will be three interpretation statements:

(6) (a) $\mathcal{D}_{D,e} = D$

(b) $\mathcal{D}_{D,t} = \{0, 1\}$

(c) $\mathcal{D}_{D,\sigma\to\tau} = \mathcal{D}_{D,\tau}^{\mathcal{D}_{D,\sigma}}$

The interpretation domain of intransitive verbs, for example, is the set of functions $\mathcal{D}_t^{\mathcal{D}_e}$. That is, $\{0, 1\}^D$. And a two-place relation such as *love* is the set of functions mapping entities onto sets of entities, $(\mathcal{D}_t^{\mathcal{D}_e})^{\mathcal{D}_e}$, in other words, $(\{0, 1\}^D)^D$.

A model for a type language consists of D and an interpretation function I. As with our model of first order logic, we will use I as the interpretation function, and instead of the assignment function U we will use g to indicate assignments. The interpretation function $[\![\ ]\!]_{\mathbf{M},g}$ is a function which maps a well formed expression into $\mathcal{D}_{D,\sigma}$. So, for an expression α, we can define the interpretations as follows:

(7) (a) If $\alpha \in \text{CON}_\sigma^L$, then $[\![\alpha]\!]_{\mathbf{M},g} = I(\alpha)$
If $\alpha \in \text{VAR}_\sigma$, then $[\![\alpha]\!]_{\mathbf{M},g} = g(\alpha)$

(b) If $\alpha \in \text{WE}_{\sigma\to\tau}^L$, and $\beta \in \text{WE}_\sigma^L$, then $[\![\alpha(\beta)]\!]_{\mathbf{M},g} = [\![\alpha]\!]_{\mathbf{M},g}\ ([\![\beta]\!]_{\mathbf{M},g})$.

(c) If $\phi, \psi \in \text{WE}_t^L$, then
$[\![\neg\phi]\!]_{\mathbf{M},g} = 1$ iff $[\![\phi]\!]_{\mathbf{M},g} = 0$
$[\![\phi \wedge \psi]\!]_{\mathbf{M},g} = 1$ iff $[\![\phi]\!]_{\mathbf{M},g} = 1$ and $[\![\psi]\!]_{\mathbf{M},g} = 1$
$[\![\phi \vee \psi]\!]_{\mathbf{M},g} = 1$ iff $[\![\phi]\!]_{\mathbf{M},g} = 1$ or $[\![\psi]\!]_{\mathbf{M},g} = 1$
$[\![\phi \to \psi]\!]_{\mathbf{M},g} = 0$ iff $[\![\phi]\!]_{\mathbf{M},g} = 1$ and $[\![\psi]\!]_{\mathbf{M},g} = 0$
$[\![\phi \leftrightarrow \psi]\!]_{\mathbf{M},g} = 1$ iff $[\![\phi]\!]_{\mathbf{M},g} = [\![\psi]\!]_{\mathbf{M},g}$

(d) If $\phi \in \text{WE}_t^L$, $v \in \text{VAR}_\sigma$, then
$[\![\forall v\phi]\!]_{\mathbf{M},g} = 1$ iff for all $d \in \mathcal{D}_{D,\sigma}$: $[\![\phi]\!]_{\mathbf{M},g[v/d]} = 1$
$[\![\exists v\phi]\!]_{\mathbf{M},g} = 1$ iff there is at least one $d \in \mathcal{D}_{D,\sigma}$: $[\![\phi]\!]_{\mathbf{M},g[v/d]} = 1$

(e) If $\phi \in \text{WE}_t^L$, $v \in \text{VAR}_\sigma$, then
$\lambda v\phi \in \text{WE}_{\sigma\to\tau}^L$;
and if $\beta \in \text{WE}_\sigma^L$, then $[\![\lambda v\phi(\beta)]\!]_{\mathbf{M},g} = [\![\lambda v\phi]\!]_{\mathbf{M},g}\ ([\![\beta]\!]_{\mathbf{M},g})$.

(f) If $\alpha, \beta \in \text{WE}_\sigma^L$, then $[\![\alpha = \beta]\!]_{\mathbf{M},g} = 1$ iff $[\![\alpha]\!]_{\mathbf{M},g} = [\![\beta]\!]_{\mathbf{M},g}$

3 The Type of a Quantified Expression

In Section 1.0, we only considered NPs that were proper names, hence interpreted as constants in our model. But we need to be able to refer to individuals by description, and not by name. For example, we need some way to talk about the following NPs: *a man*, *every course*, *some professor*. We sort of know what we want the formula to look like when they're combined with predicates:

(8) a. a man walked. $\exists x[man(x) \wedge walk(x)]$.
So, "a man" contains $\exists x[man(x) \wedge ?\ ]$

b. every course is full. $\forall x[course(x) \rightarrow full(x)]$.
So, "every course" contains $\forall x[course(x) \rightarrow ?\ ]$

c. some professor taught. $\exists x[professor(x) \wedge taught(x)]$.
So, "some professor" contains $\exists x[professor(x) \wedge ?\ ]$

Now, what is the type for a quantified NP such as *a man*? Well, look at (a) above. If the predicate *walk* is typed $e \rightarrow t$, then we could say that this quantified NP is looking for a predicate to make a proposition: i.e., it's looking for $e \rightarrow t$ to make a t. Hence, it's type is $(e \rightarrow t) \rightarrow t$.

So, now let's return to the examples above, and complete the semantic expression for each quantified NP. If we let P and Q stand for predicate variables, that is, variables of type $e \rightarrow t$, then we can finish the picture.

(9) a. *a man*: $\lambda P \exists x[man(x) \wedge P(x)]$

b. *every course*: $\lambda P \forall x[course(x) \rightarrow P(x)]$

c. *some professor*: $\lambda P \exists x[professor(x) \wedge P(x)]$

Now, let us imagine what the semantics of the quantifier all by itself is. Again, take away the predicative meaning of the noun that is the head of the NP, which is typed as $e \rightarrow t$. Then we can represent **some**, **every**, and **the** as follows. We are essentially following the definitions from Montague (1970).

(10) (i) **a**: $\lambda P \lambda Q \exists x[P(x) \wedge Q(x)]$

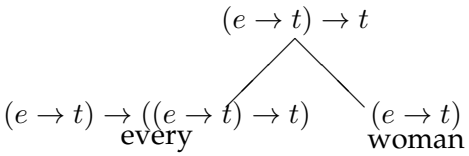
(ii) **every**: $\lambda P \lambda Q \forall x[P(x) \rightarrow Q(x)]$

(iii) **the**: $\lambda P \lambda Q \exists x \forall y[[P(y) \leftrightarrow x = y] \wedge Q(x)]$

An example of the definite NP is shown below:

(11) (a) *the teacher*: $\lambda P \exists x \forall y[[teacher(y) \leftrightarrow x = y] \wedge P(x)]$

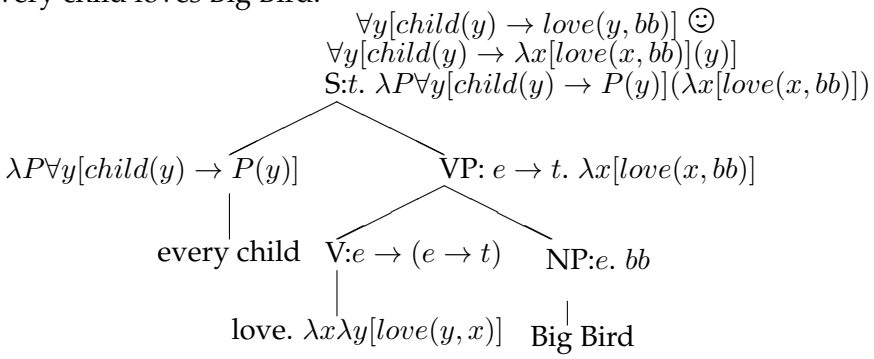
The type tree for a quantifier, such as *every*, combining with a common noun such as *woman* is illustrated below:



The resulting functional expressions are called *generalized quantifiers*.

Now let us look at how quantifiers show up in transitive verbs, such as *love*, *buy*, and *eat*. First, consider the QNP in subject position, as with the sentence below.

a. Every child loves Big Bird.

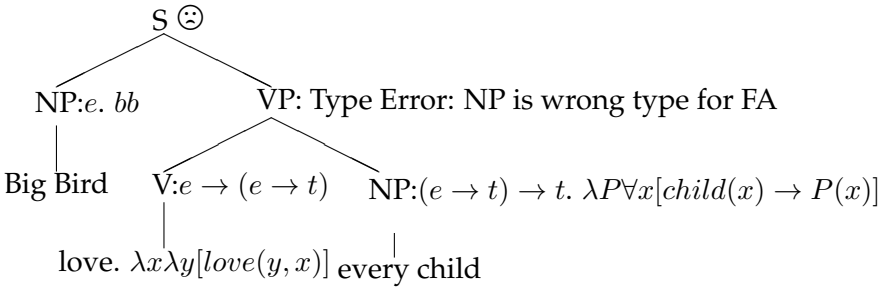


4 Quantifier Substitution

It would be nice to think that compositional mechanisms do not change that much depending on the nature of the argument. But they do, and this is nowhere more apparent than when a quantified expression appears anywhere other than in subject position of the sentence, as illustrated below.

a. Big Bird loves every child.

b.



To solve this kind of type mismatch, we're going to rework a very clever algorithm that Robin Cooper came up with, usually referred to as *Cooper Storage* (Cooper, 1975).

We will attempt to maintain the functional behavior for a predicate that we see in the syntax, as well as that which we saw above for constants in all argument positions. In other words, if a verb selects an argument that is typed as e, then it will take an argument of that type, even if the argument is a QNP such as *every woman*. I will refer to this as a *Quantifier Substitution(QS)*, and informally it looks like the following.

(12) a. Big Bird loves every child. $\Longrightarrow$

b. Big Bird loves C. $\{[C/every\ child]_{\sigma}\}$.

This says that there is a substitution, σ, where the semantics for "every child" is replace by the constant, C. Notice that it is very similar to the quantifier elimination rules in Chapter 2 of the Semantics book, where we can infer an arbitrary constant, a, in place of a quantifier scoping over a formula. Then, once the function application semantics of the entire expression has been worked through, we substitute the two expressions, which then results in further function application possibilities.

(13) Quantifier Substitution:
For every expression, γ, in a sentence, we associate a body, α, and the set of quantifier substitutions, Σ, where $\Sigma = \{[C_1/Q_1]_{\sigma_1}, [C_2/Q_2]_{\sigma_2}, \ldots, [C_n/Q_n]_{\sigma_n}\}$
$\gamma = \alpha\{\Sigma\}$

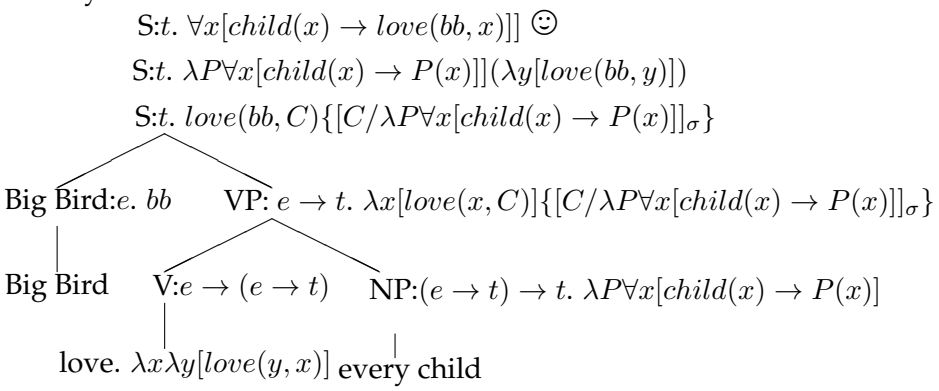
We now define a rule called *Substitution Application*. This applies to each substitution, σ_i in Σ, and it performs the following operation:

(14) $\alpha\{\sigma_u\} \Longrightarrow \sigma_u(\lambda u \alpha[u])$

Let's give a simple example to see how this works. We return to the derivation that broke earlier, namely, "Big Bird loves every child".

(15) STEP-BY-STEP:
a. Big Bird loves every child.
b. loves: $\lambda x \lambda y[love(y, x)]$
c. every child: $\lambda P \forall x[child(x) \rightarrow P(x)]$
d. Quantifier Substitution (QS): $C : e, [C/\lambda P \forall x[child(x) \rightarrow P(x)]]_\sigma$
e. Function Application: $\lambda x \lambda y[love(y, x)] : e \rightarrow (e \rightarrow t), C : e \Longrightarrow$
$\lambda y[love(y, C)] : e \rightarrow t$
f. VP now denotes: $\lambda y[love(y, C)]\{\sigma\}$
g. Function Application: $\lambda y[love(y, C)]\{\sigma\}(bb) \Longrightarrow$
$[love(bb, C)]\{\sigma\}$
h. Substitution Application: $[love(bb, C)]\{\sigma\} \Longrightarrow$
$\lambda P \forall x[child(x) \rightarrow P(x)](\lambda y[love(bb, y)])$
i. Function Application: $\lambda P \forall x[child(x) \rightarrow P(x)](\lambda y[love(bb, y)]) \Longrightarrow$
$\forall x[child(x) \rightarrow \lambda y[love(bb, y)](x)]$
j. Function Application: $\forall x[child(x) \rightarrow \lambda y[love(bb, y)](x)] \Longrightarrow$
$\forall x[child(x) \rightarrow love(bb, x)]$
k. ☺

(16) TREE DERIVATION:
a. Big Bird loves every child.
b.



If the quantifier substitution operation is similar to *quantifier elimination*, then the substitution application is similar to a successful *arrow introduction*, illustrating that the type shifting of the QNP to an individual was successful.
This technique also works for embedded quantiers and coordinate NP constructions, which we will get to shortly. We will get to this topic in the next handout.