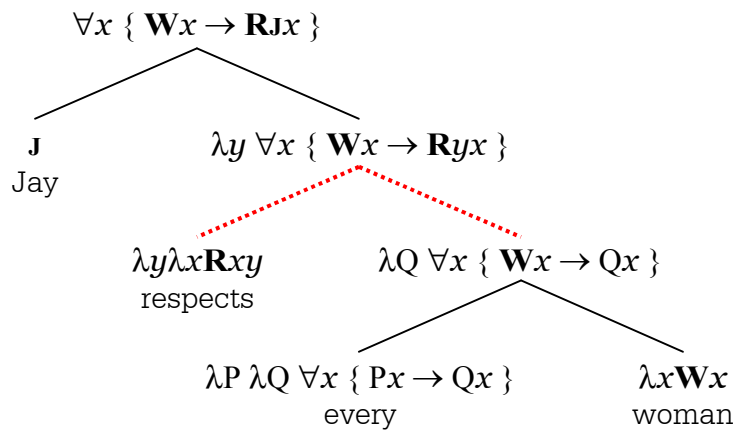


### 1. Standard Categorical-Composition

Two meanings combine precisely if one serves as an argument for the other, in which case the combined meaning is obtained by applying the function to the argument.

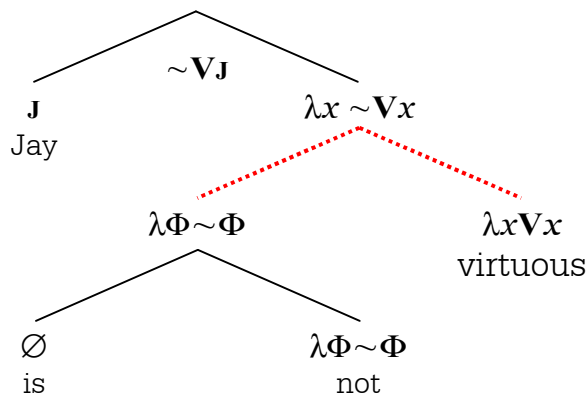
### 2. Counter-Examples

- Jay respects every woman



In the above tree, [[respects]] cannot be combined with [[every woman]] by standard-composition, since neither phrase serves as a type-appropriate argument for the other.

- Jay is not virtuous



In the above tree, even supposing that [[is]] and [[not]] compose as suggested, [[is not]] and [[virtuous]] do not compose properly, since neither phrase serves as a type-appropriate argument for the other.

### 3. Expanded Categorical Semantics

Just as we did in categorial syntax, we enlarge the permitted compositions in accordance with categorial logic. Note, for the sake of simplicity, for the moment, we consider only binary compositions.

$\mathcal{E}_1$  composes with  $\mathcal{E}_2$  to produce  $\mathcal{E}_3$  precisely if  $\mathcal{E}_1 ; \mathcal{E}_2 \vdash \mathcal{E}_3$   
 which is to say that there is a derivation of  $\mathcal{E}_3$  from  $\{\mathcal{E}_1, \mathcal{E}_2\}$  in the associated **categorial logic**.

## 4. Categorical Logic – Semantic Version

### 1. Definition of Semantic-Derivation

A semantic-derivation of  $\mathcal{E}_3$  from  $\{\mathcal{E}_1, \mathcal{E}_2\}$  is a sequence of lines as follows.

line number	expression	type	index	annotation
$\ell_1$	$\mathcal{E}_1$	$\mathfrak{J}(\mathcal{E}_1)$	$m_1$	Pr
$\ell_2$	$\mathcal{E}_2$	$\mathfrak{J}(\mathcal{E}_2)$	$m_2$	Pr
...	...	...	...	...
$\ell_3$	$\mathcal{E}_3$	$\mathfrak{J}(\mathcal{E}_3)$	$m_1, m_2$	...

In particular:

- (1) the first two lines are the premises –  $\mathcal{E}_1, \mathcal{E}_2$ .
- (2) every remaining line is either
  - (1) an assumption, or
  - (2) follows from previous lines by an inference-rule.
- (3) the last line is  $\mathcal{E}_3$ .

### 2. Sub-Structural (Index) Rules

Indices, which serve as sub-structural markers, are sequences of numerals. Different sub-structural logics are obtained by placing different restrictions on indices. In the case of System R (of Relevance Logic), we have the following restrictions.<sup>1</sup>

associativity	$(i+j)+k = i+(j+k)$
commutativity	$i+j = j+i$
contractivity	$i+i = i$
identity	$i+\emptyset = \emptyset$

Linear Logic abandons contractivity. System G abandons identity, and also places restrictions on rules (see below).

### 3. The Premise-Rule

line number	expression	type	index	annotation
$\ell$	$\mathcal{E}$	$\mathfrak{J}(\mathcal{E})$	$m$ (new)	Pr

Here,  $\mathcal{E}$  is any expression of the  $\lambda$ -calculus, and  $m$  is any numeral that is new, which is to say it does not occur earlier in the derivation. Generally, we write all the premises first, and number the indices starting with 1.

### 4. The Assumption-Rule

line number	expression	type	index	annotation
$\ell$	$v$ (new)	$\mathfrak{J}(v)$	$m$ (new)	As

Here,  $v$  is any variable of the  $\lambda$ -calculus that is new, which is to say it does not occur *unbound* earlier in the derivation, and  $m$  is any numeral that is new, which is to say it does not occur earlier in the derivation.

<sup>1</sup> Note that these conditions amount to saying that indices behave like sets.

## 5. Lambda-Out (l O) [arrow-out]

line number	expression	type	index	annotation	restriction
$l_1$	$\Lambda$	$\mathcal{A} \rightarrow \mathcal{C}$	$i$	...	$i \not\subseteq j$ and $j \not\subseteq i$
$l_2$	$\alpha$	$\mathcal{A}$	$j$	...	
$l_3$	$[\Phi]\langle\alpha\rangle$	$\mathcal{C}$	$i+j$	$l_1, l_2, \lambda O$	

Here,  $[\Lambda]\langle\mathcal{E}\rangle$  is the result of applying functor  $\Lambda$  to argument  $\alpha$ . Here,  $i$  and  $j$  are indices (sequences of numerals), and  $i+j$  is the **sequential-sum** of  $i$  and  $j$ . [For example  $\langle 1,2,3\rangle + \langle 2,4,5\rangle = \langle 1,2,3,2,4,5\rangle$ .]

## 6. Lambda-In (l I) [arrow-in]

line number	expression	type	index	annotation
$l_1$	$v$	$\mathcal{A}$	$j$	...
$l_2$	$\mathcal{E}$	$\mathcal{C}$	$i+j$	...
$l_3$	$\lambda v\{\mathcal{E}\}$	$\mathcal{A} \rightarrow \mathcal{C}$	$i$	$l_1, l_2, \lambda I$

## 7. Lambda-Calculus

At any point in a derivation, one can apply any *equivalence* of the lambda-calculus to any expression. This includes most prominently applications of **lambda-conversion**.

## 5. Examples

### 1. Transitivity

The following is an example of a valid composition, in which A, B, and C are arbitrary types, and we (temporarily) assume that the letters introduced in lines (1)-(3) have the types specified.

$$B \rightarrow C ; A \rightarrow B \vdash A \rightarrow C$$

(1)	<b>P</b>	$B \rightarrow C$	1	Pr
(2)	<b>Q</b>	$A \rightarrow B$	2	Pr
(3)	$x$	$A$	3	As
(4)	$[\mathbf{Q}]\langle x \rangle$	$B$	23	$2,3,\lambda O$
(5)	$[\mathbf{P}]\langle [\mathbf{Q}]\langle x \rangle \rangle$	$C$	123	$1,4,\lambda O$
(6)	$\lambda x [\mathbf{P}]\langle [\mathbf{Q}]\langle x \rangle \rangle$	$A \rightarrow C$	12	$3,5,\lambda I$

The following is an example of transitivity in which we compose  $\llbracket (\text{is not}) \rrbracket$  and  $\llbracket \text{virtuous} \rrbracket$  to obtain  $\llbracket (\text{is not virtuous}) \rrbracket$ .

(1)	$\lambda \Phi \sim \Phi$	$\llbracket (\text{is not}) \rrbracket$	$S \rightarrow S$	1	Pr
(2)	$\lambda x \mathbf{V}x$	$\llbracket \text{virtuous} \rrbracket$	$D \rightarrow S$	2	Pr
(3)	$x$	$D$	$D$	3	As
(4)	$\mathbf{V}x$	$S$	$S$	23	$2,3,\lambda O,\lambda C$
(5)	$\sim \mathbf{V}x$	$S$	$S$	123	$1,4,\lambda O,\lambda C$
(6)	$\lambda x \sim \mathbf{V}x$	$\llbracket (\text{is not virtuous}) \rrbracket$	$D \rightarrow S$	12	$3,5,\lambda I$

## 2. QPs in Object-Position

Earlier, we were unable to combine  $\llbracket \text{respects} \rrbracket$  and  $\llbracket \text{every woman} \rrbracket$  because of a type mismatch. The following derivation demonstrates how the appropriate meaning may be obtained using categorial logic.

$$\lambda y \lambda x \mathbf{R}xy ; \lambda Q \forall y \{ \mathbf{W}y \rightarrow Qy \} \vdash \lambda x \forall y \{ \mathbf{W}y \rightarrow \mathbf{R}xy \}$$

(1)	$\lambda z \lambda x \mathbf{R}xy$ $\llbracket \text{respects} \rrbracket$	$D \rightarrow .D \rightarrow S$	1	Pr
(2)	$\lambda Q \forall y \{ \mathbf{W}y \rightarrow Qy \}$ $\llbracket \text{every woman} \rrbracket$	$D \rightarrow S. \rightarrow S$	2	Pr
(3)	$x$	$D$	3	As
(4)	$z$	$D$	4	As
(5)	$[\lambda y \lambda x \mathbf{R}xy] \langle z \rangle$	$D \rightarrow S$	14	1,4, $\lambda O$
	$\lambda x \mathbf{R}xz$			$\lambda C$
(6)	$[\lambda x \mathbf{R}xz] \langle x \rangle$	$S$	134	3,5, $\lambda O$
	$\mathbf{R}xz$			$\lambda C$
(7)	$\lambda z \mathbf{R}xz$	$D \rightarrow S$	13	4,6, $\lambda I$
(8)	$[\lambda Q \forall y \{ \mathbf{W}y \rightarrow Qy \}] \langle \lambda z \mathbf{R}xz \rangle$	$S$	123	2,7, $\lambda O$
	$\forall y \{ \mathbf{W}y \rightarrow [\lambda z \mathbf{R}xz] \langle y \rangle \}$			$\lambda C$
	$\forall y \{ \mathbf{W}y \rightarrow \mathbf{R}xy \}$			$\lambda C$
(9)	$\lambda x \forall y \{ \mathbf{W}y \rightarrow \mathbf{R}xy \}$ $\llbracket \text{R's every W} \rrbracket$	$D \rightarrow S$	12	3,8, $\lambda I$

## 6. Fatal Flaw

Unfortunately, the following *shorter* derivation is also available.

$$\lambda y \lambda x \mathbf{R}xy ; \lambda Q \forall y \{ \mathbf{W}y \rightarrow Qy \} \vdash \lambda x \forall y \{ \mathbf{W}y \rightarrow \mathbf{R}yx \}$$

(1)	$\lambda y \lambda x \mathbf{R}xy$ $\llbracket \text{respects} \rrbracket$	$D \rightarrow .D \rightarrow S$	1	Pr
(2)	$\lambda Q \forall y \{ \mathbf{W}y \rightarrow Qy \}$ $\llbracket \text{every woman} \rrbracket$	$D \rightarrow S. \rightarrow S$	2	Pr
(3)	$x$	$D$	3	As
(4)	$[\lambda y \lambda x \mathbf{R}xy] \langle x \rangle$	$D \rightarrow S$	13	1,3, $\lambda O$
	$[\lambda y \lambda z \mathbf{R}zy] \langle x \rangle$			AV *
	$\lambda z \mathbf{R}zx$			$\lambda C$
(5)	$[\lambda Q \forall y \{ \mathbf{W}y \rightarrow Qy \}] \langle \lambda z \mathbf{R}zx \rangle$	$S$	123	2,4, $\lambda O$
	$\forall y \{ \mathbf{W}y \rightarrow [\lambda z \mathbf{R}zx] \langle y \rangle \}$			$\lambda C$
	$\forall y \{ \mathbf{W}y \rightarrow \mathbf{R}yx \}$			$\lambda C$
(6)	$\lambda x \forall y \{ \mathbf{W}y \rightarrow \mathbf{R}yx \}$ <b>!?!</b> $\llbracket \text{R's every W} \rrbracket$	$D \rightarrow S$	12	3,5, $\lambda I$

\*AV is **alphabetic variance**, which is standard fare in elementary logic, and which enables the permutation of bound ("dummy") variables. It is crucial in this computation, since otherwise lambda-conversion cannot be applied [because of the "free for" restriction].

Thus, as it stands categorial logic predicts that one reading of 'Jay respects every woman' is *that every woman respects Jay!* This is a

**complete disaster!**

## 7. Case-Markers to the Rescue

When we combine ‘respects’ with ‘every woman’, we need to know whether ‘every woman’ is the subject or the object of the verb. This is exactly what case-markers are designed to distinguish. We have already incorporated case-markers into categorial syntax. We now examine how to incorporate them into categorial semantics.

In order to do this, we must expand the lambda-calculus to include expressions that have case-inflected types. We must also expand our categorial logic to include case-inflected lambda expressions.

## 8. Case-Inflected Type-Theory

### 1. Overview

We propose to expand basic type-theory in three ways:

- (1) we expand the definition of types so that  $\times$  is a full-fledged type-operator, and we correspondingly add  $\times$  to the formation rules;
- (2) we add case-markers and case-inflection;
- (3) we expand lambda-abstraction so that the input expression can be *any* open expression.

### 2. Types

#### 1. Primitive types

- (1) D is a primitive type; [definite-noun-phrases]
- (2) S is a primitive type; [sentences]
- (3) nothing else is a primitive type.

#### 2. Case-Markers

- (1) every integer is a case-marker;
- (2) nothing else is a case-marker.

#### 3. (Case-)Marked Types

- (1) if  $\theta$  is a case-marker, then  $D_\theta$  is a (case-)marked type;
- (2) nothing else is a (case-)marked type.<sup>2</sup>

#### 4. Types

- (1) every primitive type is a type;
- (2) every marked type is a type;
- (3) if  $\mathcal{A}$  and  $\mathcal{B}$  are types, then  $[\mathcal{A} \rightarrow \mathcal{B}]$  is a type;
- (4) if  $\mathcal{A}_1, \dots, \mathcal{A}_k$  are types,  $[\mathcal{A}_1 \times \dots \times \mathcal{A}_k]$  is a type;<sup>3</sup>
- (5) nothing else is an type.

<sup>2</sup> We could expand case-marking to other types, but for the moment we concentrate on this limited application.

<sup>3</sup> Notice that  $\times$  is officially an *anadic* operator, spelled by writing a copy of it between every adjacent pair of arguments, which makes it look like a *dyadic* operator, except that there are no internal parentheses.

### 3. Formation Rules

#### 1. Variables

For each **non-marked** type  $\mathfrak{I}$ , there is an infinite list of variables of type  $\mathfrak{I}$ .<sup>4</sup>

#### 2. Multiplication

If  $\mathcal{E}_1, \dots, \mathcal{E}_k$  are expressions of types  $\mathfrak{I}_1, \dots, \mathfrak{I}_k$  respectively, then

$$[\mathcal{E}_1 \times \dots \times \mathcal{E}_k]$$

is an expression of type  $[\mathfrak{I}_1 \times \dots \times \mathfrak{I}_k]$ .

Note: since  $\mathfrak{I}_1 \times \dots \times \mathfrak{I}_k$  is the  $k$ -fold Cartesian product of  $\mathfrak{I}_1, \dots, \mathfrak{I}_k$ , a cross-product of items basically amounts to an ordered  $k$ -tuple or sequence of those items.

#### 3. Case-Inflection – Basic Scheme

If  $\mathcal{E}$  is an expression of type  $D$ , and  $\theta$  is a case-marker, then

$$\mathcal{E}_\theta$$

is an expression of type  $D_\theta$ .

#### 4. Lambda-Abstraction (Hugely Expanded)

If  $\alpha$  is an open expression of type  $\mathcal{A}$ , and  $\Omega$  is an expression of type  $\mathcal{B}$ , then

$$\lambda\alpha\Omega$$

is an expression of type  $\mathcal{A} \rightarrow \mathcal{B}$ .<sup>5</sup>

#### 5. Case-Inflection – Expanded Notation

Note that this is very powerful – the input expression  $\mathcal{E}$  can any (open) expression, even a lambda-abstract!

This allows us to write more complex case-inflected expressions according to the following shorthand definitions.

$$F_\theta \quad =_{df} \quad \lambda v_\theta Fv$$

Here,  $F$  is any atomic functor of type  $\mathcal{A} \rightarrow \mathcal{B}$ , and  $v$  is any variable of type  $\mathcal{A}$ . The most common use of this shortcut involves one-place predicates, as in the following instance.

$$P_1 \quad =_{df} \quad \lambda x_1 Px$$

Notice that  $P_1$  has type  $D_1 \rightarrow S$ . But then we can write expressions such as the following.

$$\lambda P_1 \forall x Px$$

which is short for

$$\lambda(\lambda x_1 Px) \forall x Px$$

which has type  $[(D_1 \rightarrow S) \rightarrow S]$ .

<sup>4</sup> We don't have a dedicated class of variables for marked-types; we will rather use expanded abstraction to take care of this; in particular a *complex* expression such as ' $x_1$ ' is not technically a variable, but in effect serves as a variable ranging over case-marked entities.

<sup>5</sup> Very important point – the resulting lambda-abstract *need not* denote a *function*, although it does denote a *relation*. See later section on lambda-conversion and its restrictions.

## 6. Polyadic-Abstraction

Given the presence of product-expressions, and given the expanded nature of lambda-abstraction, we do not need to posit polyadic abstraction as a primitive. Henceforth,

$$\lambda(v^1 \times \dots \times v^k)\Omega$$

*officially* replaces

$$\lambda v^1 \dots v^k \Omega$$

although we use the latter as shorthand for the former.

## 7. Lambda-Conversion

Given how complicated the new class of lambda-abstracts are, we must revise the rule of lambda-conversion accordingly, which is done as follows.

### 1. Original Form (monadic)

$$[\lambda v \Omega] \langle \sigma \rangle = \Omega[\sigma/v]$$

Here,  $\Omega[\sigma/v]$  results from substituting  $\sigma$  for each occurrence of  $v$  in  $\Omega$  that is free in  $\Omega$ . Also  $v$  is free for  $\sigma$  in  $\Omega$ , which is to say that every variable that is free in  $\sigma$  is also free in  $\Omega[\sigma/v]$ .

### 2. Expanded Form

$$[\lambda \alpha \Omega] \langle \alpha[\sigma_1/v_1, \dots, \sigma_k/v_k] \rangle = \Omega[\sigma_1/v_1, \dots, \sigma_k/v_k]$$

Here, for each  $i$ ,  $\text{type}(\sigma_i) = \text{type}(v_i)$ , and  $v_i$  is free for  $\sigma_i$  in  $\alpha$ , and for  $\sigma_i$  in  $\Omega$ .

Also,  $\alpha[\sigma_1/v_1, \dots, \sigma_k/v_k]$  results from substituting  $\sigma_i$  for each occurrence of  $v_i$  in  $\alpha$  that is free in  $\alpha$ ; similarly for  $\Omega[\sigma_1/v_1, \dots, \sigma_k/v_k]$ .

### KEY PROVISIO:

$$\begin{array}{l} \text{for any } a_1, \dots, a_k \text{ and } b_1, \dots, b_k, \\ \text{if } \alpha[a_1/v_1, \dots, a_k/v_k] = \alpha[b_1/v_1, \dots, b_k/v_k] \\ \text{then } \Omega[a_1/v_1, \dots, a_k/v_k] = \Omega[b_1/v_1, \dots, b_k/v_k] \end{array}$$

Note, in asserting this proviso, we are saying that **lambda-conversion fails** when the proviso fails. Although the lambda-abstract is well-formed, it does not denote a function, but rather a one-many relation.<sup>6</sup>

<sup>6</sup> The following is an example, from arithmetic, that does not satisfy the condition.

$$\lambda(x^2)(x^3)$$

In particular, whereas  $(2)^2 = (-2)^2$ , but  $(2)^3 \neq (-2)^3$ . Thus the relation conveyed by this abstract is not a function, since it is one-many; in particular, 4 bears the relation to both 8 and -8.

### 3. Cancellation Principles

In light of the key proviso, and in light of our account of case-marked items, we include the following axiom of cancellation for case-marked items.

$$\text{if } \alpha_0 = \beta_0, \text{ then } \alpha = \beta$$

This is not true by form/logic alone, so it must be included as a fundamental principle, which is critical to the application of expanded lambda-conversion.

Similarly, although the following principle is not true on purely formal grounds, it is plausible, since  $\times$  corresponds to sequence formation, and so we postulate it as part of expanded type theory.

$$\text{if } [\alpha_1 \times \dots \times \alpha_k] = [\beta_1 \times \dots \times \beta_k], \text{ then } [\alpha_1 = \beta_1] \ \& \ \dots \ \& \ [\alpha_k = \beta_k]$$

This is similarly critical to the application of expanded lambda-conversion.

## 9. Derivation Rules in Categorical Logic

Since we now have an expanded class of lambda-abstracts, and we also now have multiplication, we must correspondingly expand our account of semantic-composition to include these new syntactic items.

### 1. Lambda-Out (l O)

$l_1$	$\Lambda$	$\mathcal{A} \rightarrow \mathcal{C}$	$i$	...	$i \not\subseteq j$ and $j \not\subseteq i$
$l_2$	$\alpha$	$\mathcal{A}$	$j$	...	
$l_3$	$[\Lambda]\langle\alpha\rangle$	$\mathcal{C}$	$i+j$	$l_1, l_2, \lambda O$	

Here,  $[\Lambda]\langle\alpha\rangle$  is the result of combining functor  $\Lambda$  with argument  $\alpha$ , in accordance with (*expanded*) lambda-conversion.

### 2. Lambda-In (l I)

$l_1$	$\alpha$	$\mathcal{A}$	$i$	...
$l_2$	$\Omega$	$\mathcal{B}$	$i+j$	...
$l_3$	$\lambda\alpha\{\Omega\}$	$\mathcal{A} \rightarrow \mathcal{B}$	$i$	$l_1, l_2, \lambda I$

### 3. Cross-Out ( $\times O$ ) (binary version)

$l_0$	$\mathcal{E}_1 \times \mathcal{E}_2$	$\mathcal{A} \times \mathcal{B}$	$h$	...
$l_1$	$\mathcal{E}_1$	$\mathcal{A}$	$i$	...
$l_2$	$\mathcal{E}_2$	$\mathcal{B}$	$j$	...
$l_3$	$\mathcal{E}_3$	$\mathcal{B}$	$i+j+k$	...
$l_4$	$\mathcal{E}_3$	$\mathcal{C}$	$h+k$	$l_0 \ l_1-l_3, \times O$

### 4. Cross-In ( $\times I$ ) (binary version)

$l_1$	$\mathcal{E}_1$	$\mathcal{A}$	$i$	...	$i \not\subseteq j$ and $j \not\subseteq i$
$l_2$	$\mathcal{E}_2$	$\mathcal{B}$	$j$	...	
$l_3$	$\mathcal{E}_1 \times \mathcal{E}_2$	$\mathcal{A} \times \mathcal{B}$	$i+j$	$l_1, l_2, \times I$	