

# Effectful composition in natural language semantics

Effect-driven interpretation

Dylan Bumford (UCLA)   Simon Charlow (Yale)

NASSLLI 2025 @ UW

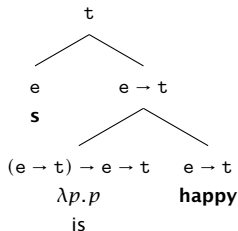
June 24, 2025

## Pronouns and effects

## Pronouns as identity maps

Jacobson (1999) begins with the idea that pronouns are **identity functions**

We've seen identity functions used when there is nothing semantically interesting to say about some bit of syntax



This is not what Jacobson has in mind: she is thinking of the pronoun as making a request for an antecedent, and then *yielding that antecedent* for composition

## Pronouns as effects

In fact, Jacobson invents a notation to distinguish between two kinds of functions:

- ▶  $\boxed{\alpha/e}$  is an ordinary function that takes  $e$  as argument and returns  $\beta$  as result (what we normally write as  $\alpha \rightarrow \beta$ )
- ▶  $\boxed{\alpha^e}$  is something that acts like a  $\alpha$ , though it is still missing an  $e$ -sized bit of information

From our perspective, we might construe the second kind of type as corresponding to a particular kind of **computational effect**: a request to the context for an antecedent:

$$R\alpha ::= e \rightarrow \alpha$$

**Ordinary function:**

$$\llbracket \text{to} \rrbracket := \underbrace{\lambda x. x}_{e \rightarrow e}$$

**Antecedent request:**

$$\llbracket \text{she} \rrbracket := \underbrace{\lambda x. x}_{R e}$$

## Composing around pronouns

$$R\alpha ::= e \rightarrow \alpha$$

$$\llbracket \text{she} \rrbracket := \underbrace{\lambda x. x}_{Re}$$

How should these compose with things like transitive verbs, which are looking for an individual, not a function from individuals to individuals?

**Jacobson's approach:** Context-sensitive composition, on demand

$$(13) \quad \mathbf{g}_c(B/A) = B^c/A^c$$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_c(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_c(f) = \lambda V[\lambda C[f(V(C))]]$ .

## Composing around pronouns

$$\mathbb{R}\alpha ::= e \rightarrow \alpha$$

$$\llbracket \text{she} \rrbracket := \underbrace{\lambda x. x}_{\mathbb{R}e}$$

How should these compose with things like transitive verbs, which are looking for an individual, not a function from individuals to individuals?

**Jacobson's approach:** Context-sensitive composition, on demand

$$(13) \quad \mathbf{g}_c(B/A) = B^c/A^c$$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_c(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_c(f) = \lambda V[\lambda C[f(V(C))]]$ .

Or in our notation:

$$\mathbf{g} :: (\alpha \rightarrow \beta) \rightarrow \mathbb{R}\alpha \rightarrow \mathbb{R}\beta$$

Sound familiar?

$$\mathbf{g} := \lambda f \lambda m \lambda r. f (m r)$$

## Pronouns as functors

Of course, this is just the  $\bullet$  for the `R` functor

```
data Reader r a = Reader (r -> a)
instance Functor (Reader r) where
  -- fmap :: (a -> b) -> Reader a -> Reader b
  fmap k (Reader m) = \x -> Reader (k (m x))
```

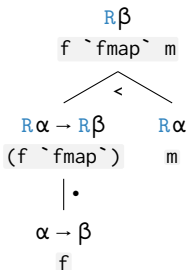
$$R\alpha ::= e \rightarrow \alpha$$
$$k \bullet m := \lambda e. k (m e)$$

## Composition with functors

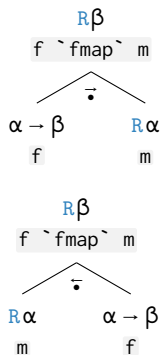
## Two approaches to composition

Then there are two ways we might imagine incorporating the map into our picture of composition:

- ▶ as a **unary combinator**  
(aka, a coercion or type-shifter):



- ▶ or as pair of **binary combinators**  
(aka, modes of combination):



# Examples

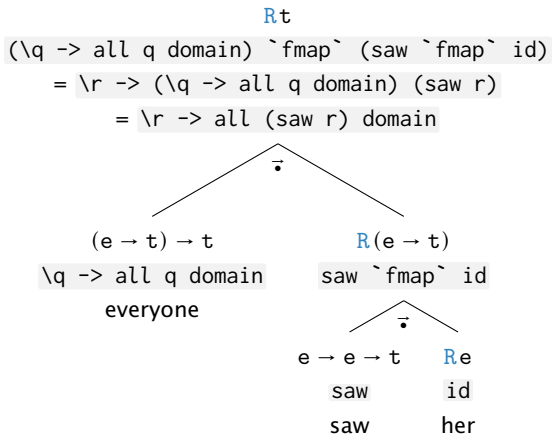
(1) She left

$$\begin{aligned} & \text{Rt} \\ & \text{left' `fmap` id} \\ = & \text{\r -> left' (id r)} \\ = & \text{\r -> left' r} \end{aligned}$$

Re	e -> t
id	left'
she	left

## Examples

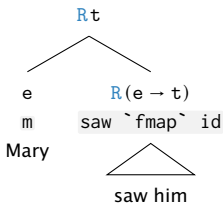
(2) Everyone saw her





## Boring subjects

Remember that  $(\bullet)$  combines an ordinary function  $k :: \alpha \rightarrow \beta$  with a fancy argument  $m :: R\alpha$ , but here we have the opposite situation



### Modes of Combination

$$(>) : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

$$f > x := f x$$

$$(<) : \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$$

$$x < f := f x$$

$$(\sqcap) : (\alpha \rightarrow t) \rightarrow (\alpha \rightarrow t) \rightarrow \alpha \rightarrow t$$

$$f \sqcap g := \lambda x. f x \wedge g x$$

$$(\vec{\bullet}) : (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

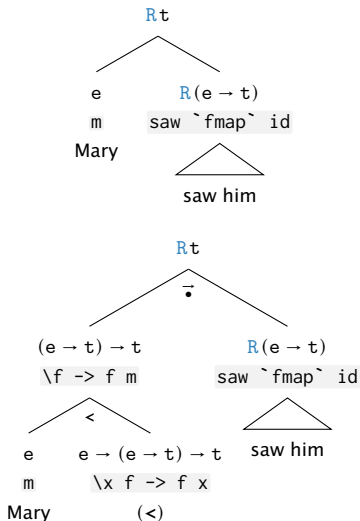
$$f \vec{\bullet} x := \lambda i. f (xi)$$

$$(\overleftarrow{\bullet}) : R\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow R\beta$$

$$x \overleftarrow{\bullet} f := \lambda i. f (xi)$$

## Boring subjects

Remember that  $(\bullet)$  combines an ordinary function  $k :: \alpha \rightarrow \beta$  with a fancy argument  $m :: R\alpha$ , but here we have the opposite situation



### Modes of Combination

$$(>) : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

$$f > x := f x$$

$$(<) : \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$$

$$x < f := f x$$

$$(\sqcap) : (\alpha \rightarrow t) \rightarrow (\alpha \rightarrow t) \rightarrow \alpha \rightarrow t$$

$$f \sqcap g := \lambda x. f x \wedge g x$$

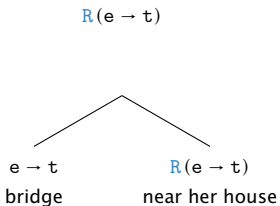
$$(\vec{\bullet}) : (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

$$f \vec{\bullet} x := \lambda i. f (x i)$$

$$(\overleftarrow{\bullet}) : R\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow R\beta$$

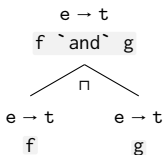
$$x \overleftarrow{\bullet} f := \lambda i. f (x i)$$

## Mapping over alternate modes



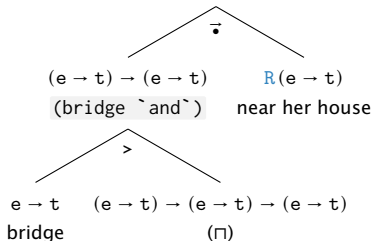
Of course what you'd like to do here is somehow use the Modification rule:

### Modification



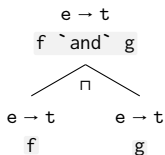
## Mapping over alternate modes

$R(e \rightarrow t)$   
`(bridge `and`) `fmap` (near `fmap` (house `fmap` id))`  
`= \r -> bridge `and` near (house r)`



Of course what you'd like to do here is somehow use the Modification rule:

### Modification



## Multiple pronouns

First question: if the meaning of (3) is...

$$(3) \begin{aligned} \llbracket \text{she left} \rrbracket &:: R t \\ \llbracket \text{she left} \rrbracket &= \lambda x. \mathbf{leave} x \end{aligned}$$

... what do you think the meaning of (4) ought to be?

(4) she saw her

## Multiple pronouns

First question: if the meaning of (3) is...

$$(3) \begin{aligned} \llbracket \text{she left} \rrbracket &:: \mathbf{R}t \\ \llbracket \text{she left} \rrbracket &= \lambda x. \mathbf{leave} x \end{aligned}$$

... what do you think the meaning of (4) ought to be?

(4) she saw her

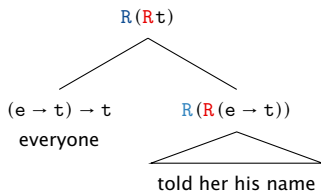
Two pronouns = two requests!

$$\begin{aligned} \text{she saw her} &:: \mathbf{R}(\mathbf{R}t) \\ \llbracket \text{she saw her} \rrbracket &= \lambda x \lambda y. \mathbf{saw} x y \end{aligned}$$

Very tricky to see how this can be derived, but let's solve a very slightly easier problem first

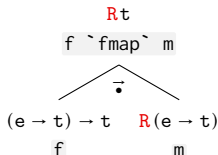
## Double skipping

Assuming we can compose the constituent with the two pronouns to something that makes two requests, how do we combine *that* with anything else?



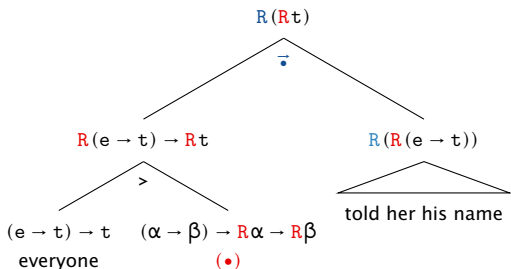
This time, what we'd really like to use is the `fmap` rule itself!

`fmap`



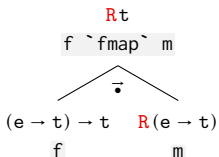
## Double skipping

Assuming we can compose the constituent with the two pronouns to something that makes two requests, how do we combine *that* with anything else?



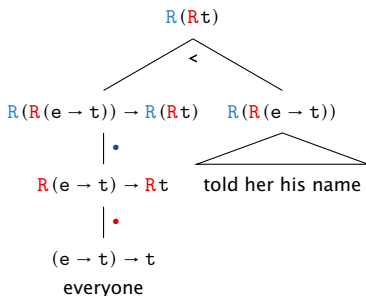
This time, what we'd really like to use is the `fmap` rule itself!

`fmap`



## Functors compose

If we write this as a composition of unary operations, it belies a rather deep fact:



The composition of two functors is always a functor

$$((f \bullet) \bullet) :: H(\Gamma \alpha) \rightarrow H(\Gamma \beta)$$
$$| \bullet$$
$$(f \bullet) :: \Gamma \alpha \rightarrow \Gamma \beta$$
$$| \bullet$$
$$f :: \alpha \rightarrow \beta$$

```
ghci> :t fmap . fmap
```

```
fmap . fmap
```

```
:: (Functor f1, Functor f2) =>
```

```
(a -> b) ->
```

```
f1 (f2 a) ->
```

```
f1 (f2 b)
```

## Now for the hard part

How do we build up the double-pronoun clause?

(29) g generalized:

a.  $\mathbf{g}_{C_0}(B/A) = B^C/A^C$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_{C_0}(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_{C_0}(f) = \lambda V[\lambda C[f(V(C))]]$ .

b.  $\mathbf{g}_{C_n}(A^B) = \mathbf{g}_{C_{n-1}}(A)^B$

If  $f$  is a function of type  $\langle d, \langle a, b \rangle \rangle$ , then  $\mathbf{g}_{C_n}(f)$  is a function of type  $\langle d, \langle \langle c, a \rangle, \langle c, b \rangle \rangle \rangle$  where  $\mathbf{g}_{C_n}(f) = \lambda D[\mathbf{g}_{C_{n-1}}(f(D))]$ .<sup>14</sup>

Let's work out the types of these in functorial terms

$$\mathbf{g}_0 = \lambda f \lambda m \lambda z. f(mz)$$

$$\mathbf{g}_0 :: (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

$$\mathbf{g}_1 = \lambda m \lambda r. \mathbf{g}_0(mr)$$

$$\mathbf{g}_i ::$$

## Now for the hard part

How do we build up the double-pronoun clause?

(29) g generalized:

a.  $\mathbf{g}_{C_0}(B/A) = B^C/A^C$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_{C_0}(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_{C_0}(f) = \lambda V[\lambda C[f(V(C))]]$ .

b.  $\mathbf{g}_{C_n}(A^B) = \mathbf{g}_{C_{n-1}}(A)^B$

If  $f$  is a function of type  $\langle d, \langle a, b \rangle \rangle$ , then  $\mathbf{g}_{C_n}(f)$  is a function of type  $\langle d, \langle \langle c, a \rangle, \langle c, b \rangle \rangle \rangle$  where  $\mathbf{g}_{C_n}(f) = \lambda D[\mathbf{g}_{C_{n-1}}(f(D))]$ .<sup>14</sup>

Let's work out the types of these in functorial terms

$$\mathbf{g}_0 = \lambda f \lambda m \lambda z. f (m z)$$

$$\mathbf{g}_0 :: (\alpha \rightarrow \beta) \rightarrow R \alpha \rightarrow R \beta$$

$$\mathbf{g}_1 = \lambda m \lambda r. \mathbf{g}_0 (\underbrace{m r}_{\alpha \rightarrow \beta})$$

$$\mathbf{g}_i ::$$

## Now for the hard part

How do we build up the double-pronoun clause?

(29) g generalized:

a.  $\mathbf{g}_{C_0}(B/A) = B^C/A^C$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_{C_0}(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_{C_0}(f) = \lambda V[\lambda C[f(V(C))]]$ .

b.  $\mathbf{g}_{C_n}(A^B) = \mathbf{g}_{C_{n-1}}(A)^B$

If  $f$  is a function of type  $\langle d, \langle a, b \rangle \rangle$ , then  $\mathbf{g}_{C_n}(f)$  is a function of type  $\langle d, \langle \langle c, a \rangle, \langle c, b \rangle \rangle \rangle$  where  $\mathbf{g}_{C_n}(f) = \lambda D[\mathbf{g}_{C_{n-1}}(f(D))]$ .<sup>14</sup>

Let's work out the types of these in functorial terms

$$\mathbf{g}_0 = \lambda f \lambda m \lambda z. f (m z)$$

$$\mathbf{g}_0 :: (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

$$\mathbf{g}_1 = \lambda \underbrace{m}_{R\alpha} \lambda r. \mathbf{g}_0 (\underbrace{m r}_{\alpha \rightarrow \beta})$$

$$\mathbf{g}_i ::$$

## Now for the hard part

How do we build up the double-pronoun clause?

(29) g generalized:

a.  $\mathbf{g}_{C_0}(B/A) = B^C/A^C$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_{C_0}(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_{C_0}(f) = \lambda V[\lambda C[f(V(C))]]$ .

b.  $\mathbf{g}_{C_n}(A^B) = \mathbf{g}_{C_{n-1}}(A)^B$

If  $f$  is a function of type  $\langle d, \langle a, b \rangle \rangle$ , then  $\mathbf{g}_{C_n}(f)$  is a function of type  $\langle d \langle \langle c, a \rangle, \langle c, b \rangle \rangle$  where  $\mathbf{g}_{C_n}(f) = \lambda D[\mathbf{g}_{C_{n-1}}(f(D))]$ .<sup>14</sup>

Let's work out the types of these in functorial terms

$$\mathbf{g}_0 = \lambda f \lambda m \lambda z. f(mz)$$

$$\mathbf{g}_0 :: (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

$$\mathbf{g}_1 = \lambda \underbrace{m}_{R\alpha} \lambda r. \mathbf{g}_0(\underbrace{mr}_{\alpha \rightarrow \beta})$$
$$R\alpha \rightarrow R\beta$$

$$\mathbf{g}_i ::$$

## Now for the hard part

How do we build up the double-pronoun clause?

(29) g generalized:

a.  $\mathbf{g}_{C^0}(B/A) = B^C/A^C$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $\mathbf{g}_{C^0}(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $\mathbf{g}_{C^0}(f) = \lambda V[\lambda C[f(V(C))]]$ .

b.  $\mathbf{g}_{C^n}(A^B) = \mathbf{g}_{C^{n-1}}(A)^B$

If  $f$  is a function of type  $\langle d, \langle a, b \rangle \rangle$ , then  $\mathbf{g}_{C^n}(f)$  is a function of type  $\langle d, \langle \langle c, a \rangle, \langle c, b \rangle \rangle \rangle$  where  $\mathbf{g}_{C^n}(f) = \lambda D[\mathbf{g}_{C^{n-1}}(f(D))]$ .<sup>14</sup>

Let's work out the types of these in functorial terms

$$\mathbf{g}_0 = \lambda f \lambda m \lambda z. f (m z)$$

$$\mathbf{g}_0 :: (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

$$\mathbf{g}_1 = \lambda \underbrace{m}_{R\alpha} \lambda r. \mathbf{g}_0 (\underbrace{m r}_{\alpha \rightarrow \beta})$$

$$\underbrace{\hspace{10em}}_{R\alpha \rightarrow R\beta}$$

$$\mathbf{g}_i ::$$

## Now for the hard part

How do we build up the double-pronoun clause?

(29) g generalized:

a.  $g_{C^0}(B/A) = B^C/A^C$

If  $f$  is a function of type  $\langle a, b \rangle$  then  $g_{C^0}(f)$  is a function of type  $\langle \langle c, a \rangle, \langle c, b \rangle \rangle$ , where  $g_{C^0}(f) = \lambda V[\lambda C[f(V(C))]]$ .

b.  $g_{C^n}(A^B) = g_{C^{n-1}}(A)^B$

If  $f$  is a function of type  $\langle d, \langle a, b \rangle \rangle$ , then  $g_{C^n}(f)$  is a function of type  $\langle d, \langle \langle c, a \rangle, \langle c, b \rangle \rangle \rangle$  where  $g_{C^n}(f) = \lambda D[g_{C^{n-1}}(f(D))]$ .<sup>14</sup>

Let's work out the types of these in functorial terms

$$g_0 = \lambda f \lambda m \lambda z. f (m z)$$

$$g_0 :: (\alpha \rightarrow \beta) \rightarrow R\alpha \rightarrow R\beta$$

$$g_1 = \lambda \underbrace{m}_{R\alpha} \lambda r. \underbrace{g_0(mr)}_{\underbrace{\alpha \rightarrow \beta}_{R\alpha \rightarrow R\beta}}$$

$$g_i :: R(\alpha \rightarrow \beta) \rightarrow R(R\alpha \rightarrow R\beta)$$

## fmap fmap

Look again at that recursive step:

$$\mathbf{g}_0 = \lambda f \lambda m \lambda z. f (m z)$$

$$\mathbf{g}_1 = \lambda m \lambda z. \mathbf{g}_0 (m z)$$

If you stare at it long enough, you may notice that  $\mathbf{g}_1 = \mathbf{g}_0 \mathbf{g}_0$

And recall that  $\mathbf{g}_0$  is the  $(\bullet)$  of `R`, which means  $\mathbf{g}_1 = (\bullet) (\bullet)$

Again, there is something quite general going on:

$$(\bullet) (\bullet) :: \mathbf{H}(\alpha \rightarrow \beta) \rightarrow \mathbf{H}(\Gamma \alpha \rightarrow \Gamma \beta)$$

|  
•

$$(\bullet) :: (\alpha \rightarrow \beta) \rightarrow \Gamma \alpha \rightarrow \Gamma \beta$$

```
ghci> :t fmap fmap
```

```
fmap fmap
```

```
:: (Functor f1, Functor f2) =>
```

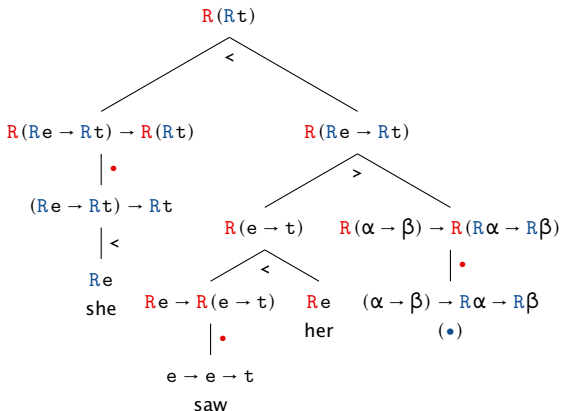
```
f1 (a -> b) ->
```

```
f1 (f2 a -> f2 b)
```

## Deriving the double pronouns: *she saw her*

(31)  $g(l_{(e,t)}(\text{his-mother'}))(g(g(\text{love'})(\text{his-dog'}))$

(NB: the second of the three  $g$ 's is  $g_1$ ; the rest are  $g_0$ )





## Comments, questions, concerns

Are silent occurrences of • actually instantiated in the syntax?

How is this *practical*? < and • can apply iteratively. How would we ever know when to stop, or that a new (or first) analysis was not around the next corner?

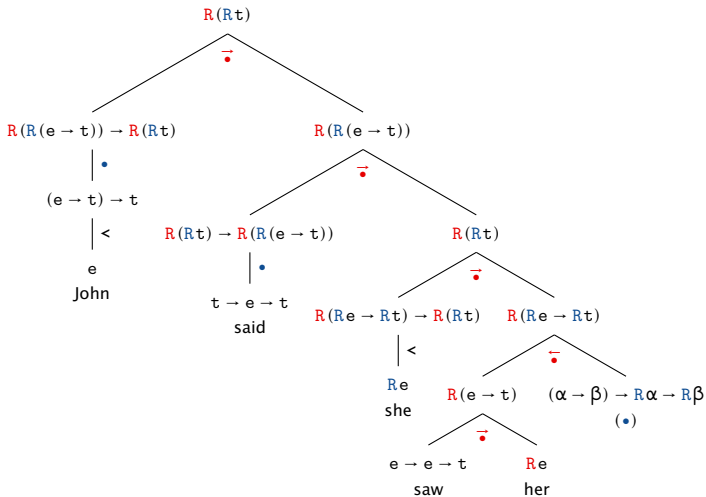
The system is unwieldy. Derivations are difficult to construct and significantly more complex than the readily justifiable syntax.

We would prefer a solution much more in the vein of **direct-style** treatments of effects in programming languages, but with referential transparency

Leveling up

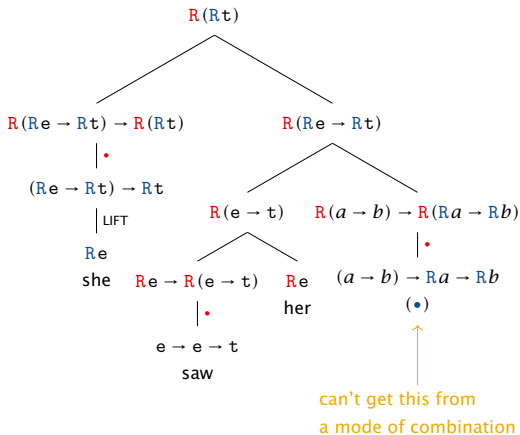
# Combinators vs. Combinations

Some of the apparent complexity boils away when we use binary **modes of combination** wherever possible

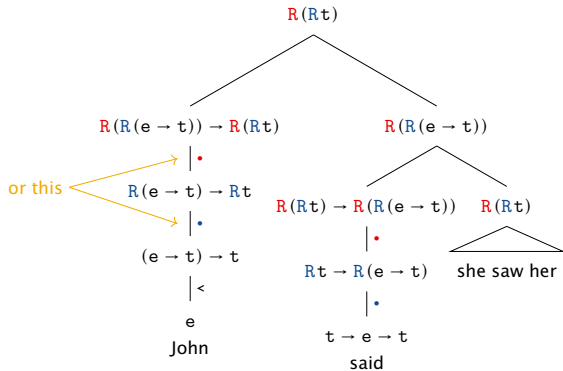


# Recursion

But much of the expressive power comes from allowing these combinators to be partially applied and/or applied to themselves



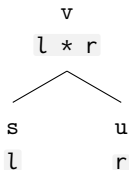
# Combinatorial recursion



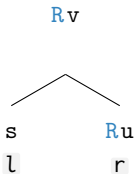
## Factoring out the patterns

## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$

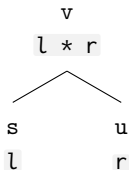


- ▶ Can you combine when the left daughter is  $s$  and the right is  $Ru$

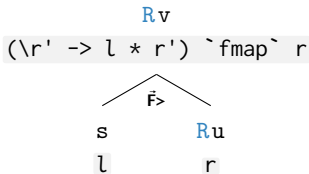


## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$

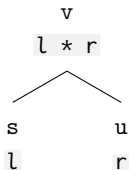


- ▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$



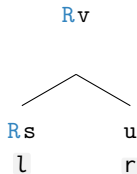
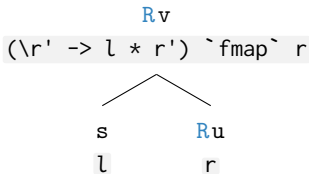
## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$



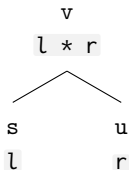
▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$

▷ What about when the left daughter is  $Rs$  and the right is  $u$



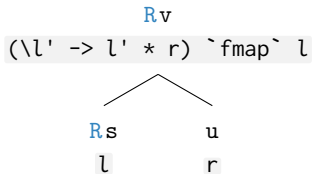
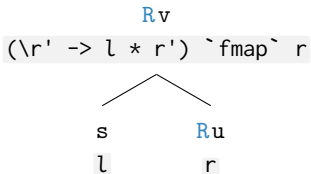
## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$



- ▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$

- ▷ What about when the left daughter is  $Rs$  and the right is  $u$

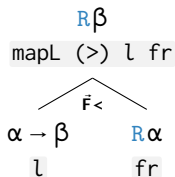
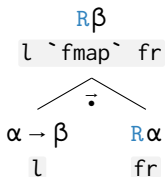


## Modes from modes

This schema gives us a way to make new modes of combination out of old ones!

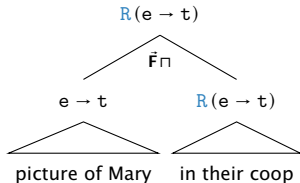
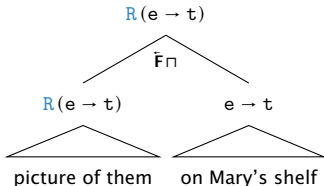
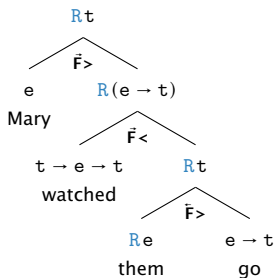
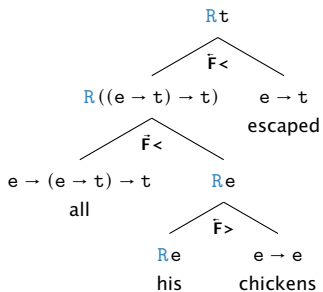
```
mapL (*) fl r = (\l -> l * r) `fmap` fl
mapR (*) l fr = (\r -> l * r) `fmap` fr
```

For instance, `mapL (>)` is equivalent to the mapping mode  $\vec{\cdot}$



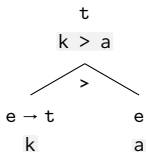
```
mapL (>) l fr = (\r -> l > r) `fmap` fr
--           = (\r -> l r) `fmap` fr
--           = l `fmap` fr
```

# In action

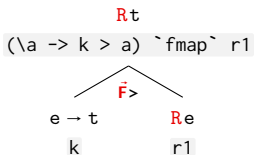


# Recurring on the right

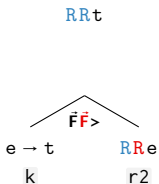
## 1st-order MoC



## 2nd-order MoC

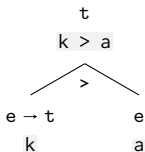


## 3rd-order MoC

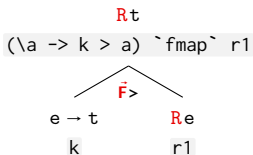


# Recurring on the right

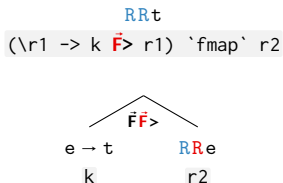
## 1st-order MoC



## 2nd-order MoC

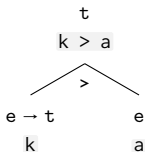


## 3rd-order MoC

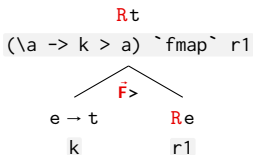


# Recurring on the right

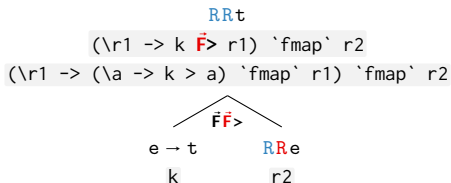
## 1st-order MoC



## 2nd-order MoC

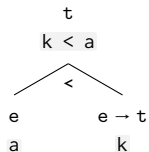


## 3rd-order MoC

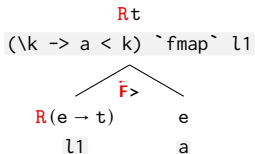


# Recurring on the left

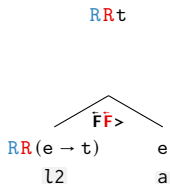
## 1st-order MoC



## 2nd-order MoC

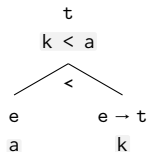


## 3rd-order MoC

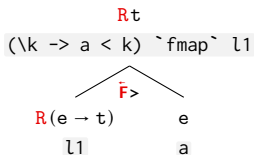


# Recurring on the left

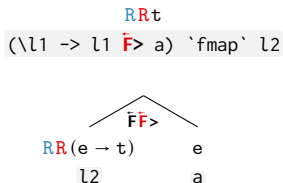
## 1st-order MoC



## 2nd-order MoC

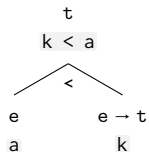


## 3rd-order MoC

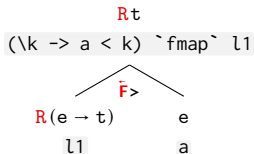


# Recurring on the left

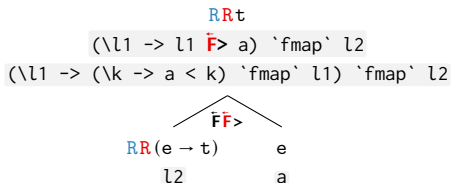
## 1st-order MoC



## 2nd-order MoC



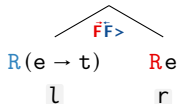
## 3rd-order MoC



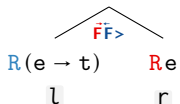
## Recurring on both sides

What happens when both daughters are functorial?

R R t



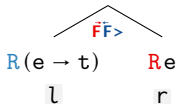
R R t



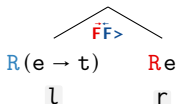
## Recurring on both sides

What happens when both daughters are functorial?

$\text{RRt}$   
`(\a -> l  $\tilde{\text{F}}$ > a) `fmap` r`

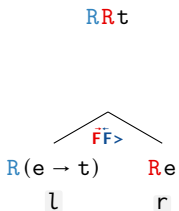
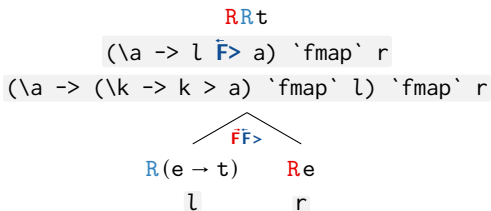


$\text{RRt}$



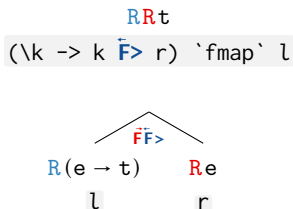
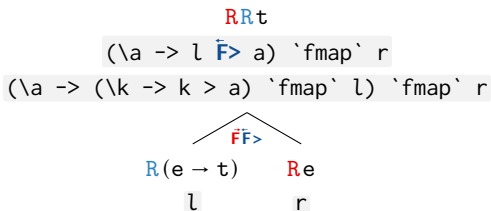
## Recurring on both sides

What happens when both daughters are functorial?



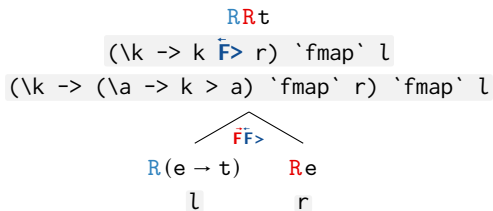
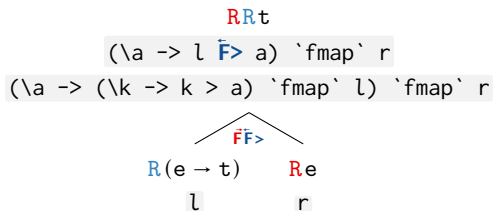
## Recurring on both sides

What happens when both daughters are functorial?



## Recurring on both sides

What happens when both daughters are functorial?



## Comments, questions, concerns addressed

Are occurrences of • actually instantiated in the syntax?

- ▶ **No!** • is in the **interpreter**, not the language

How is this *practical*? < and • can apply iteratively.

- ▶ Combination is recursive, but this recursion is pointed **down** the type hierarchy, rather than up. Semantic parsing is decidable!

The system is unwieldy. Derivations are complex and difficult to construct.

- ▶ Our semantic parses are **homomorphic** to the syntax that generates them.  
Derivations can be completely mechanized

[schar.github.io/TDParse](https://schar.github.io/TDParse)

.

Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2). 117–184.  
<https://doi.org/10.1023/A:1005464228727>.