A modular theory of pronouns (and binding)

Simon Charlow (Rutgers)

NYPLW September 25, 2017

Overview

Today: a brief on the power of **abstraction** and **modularity** in semantic theorizing, with a focus on pronouns and the grammatical mechanisms for dealing with them.

Semanticists tend to respond to things beyond the Fregean pale by lexically and compositionally generalizing to the worst case. One-size-fits-all.

Functional programmers instead look for repeated patterns, and abstract those out as separate, modular pieces (functions). When we do semantics, this strategy has conceptual and especially empirical virtues.

The standard theory, and its discontents

A baseline extensional semantic theory

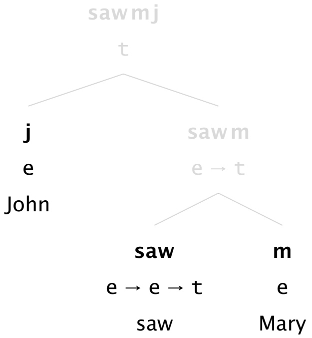
Inductively define the space of possible meanings, sorted by type:

$$\tau ::= \mathrm{e} \mid \mathrm{t} \mid \underbrace{\tau \to \tau}_{\mathrm{e}\to\mathrm{t},\ (\mathrm{e}\to\mathrm{t})\to\mathrm{t},\ \ldots}$$

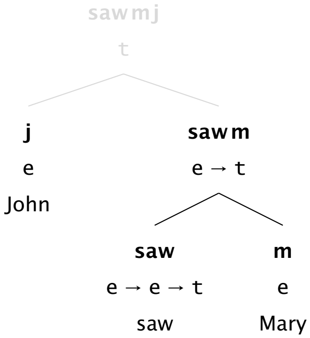
Interpret binary combination via (type-driven) functional application:

$$[\![\alpha\ \beta]\!] := [\![\alpha]\!][\![\beta]\!] \text{ or } [\![\beta]\!][\![\alpha]\!], \text{ whichever is defined}$$

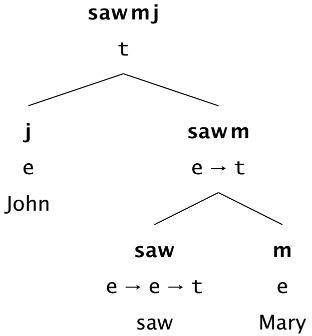
A sample derivation



A sample derivation



A sample derivation



Pronouns and binding

This picture is awesome. But a lot of important stuff doesn't fit neatly in it.

There are tons of examples, but the one I'd like to focus on today is the case of *pronouns* — how are they valued, and what ramifications does the need to value them have for the rest of the grammar?

1. John saw her$_i$.
2. Every philosopher$_i$ thinks they$_i$'re a genius.

Standardly: extending the baseline theory with assignments

Extend the baseline extensional theory such that the compositionally relevant meanings are **uniformly dependent on assignments**:

$$\tau_\circ ::= e \mid t \mid \tau_\circ \to \tau_\circ \qquad \tau ::= \underbrace{g \to \tau_\circ}_{g \to e \to t,\; g \to (e \to t) \to t,\; \ldots}$$

Interpret binary combination via **assignment-sensitive** functional application:

$$[\![\alpha\ \beta]\!] := \lambda g. [\![\alpha]\!]\, g\, ([\![\beta]\!]\, g) \text{ or } [\![\beta]\!]\, g\, ([\![\alpha]\!]\, g), \text{ whichever is defined}$$

Standardly: extending the baseline theory with assignments

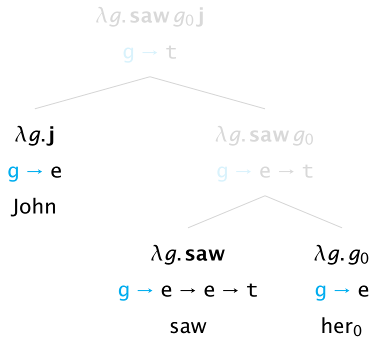
Extend the baseline extensional theory such that the compositionally relevant meanings are **uniformly dependent on assignments**:

$$\tau_\circ ::= e \mid t \mid \tau_\circ \to \tau_\circ \qquad \tau ::= \underbrace{g \to \tau_\circ}_{g\to e\to t,\ g\to(e\to t)\to t,\ \ldots}$$

Interpret binary combination via **assignment-sensitive** functional application:

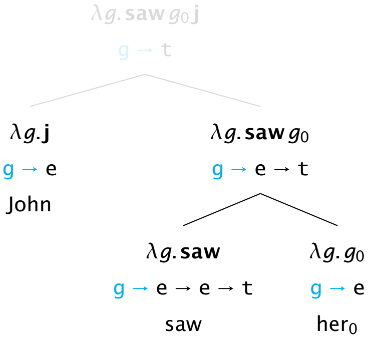
$$[\![\alpha\ \beta]\!] := \lambda g.\,[\![\alpha]\!]\,g\,([\![\beta]\!]\,g) \text{ or } [\![\beta]\!]\,g\,([\![\alpha]\!]\,g)\text{, whichever is defined}$$

Sample derivation



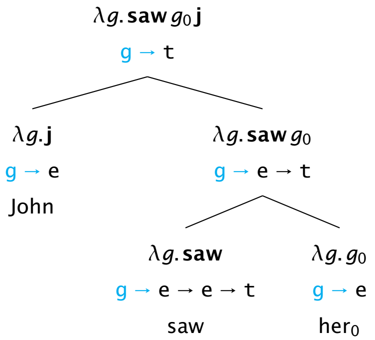
When all is said and done, apply the derived meaning to a contextual furnished assignment to derive a proposition (truth value).

Sample derivation



When all is said and done, apply the derived meaning to a contextual furnished assignment to derive a proposition (truth value).

Sample derivation



When all is said and done, apply the derived meaning to a contextual furnished assignment to derive a proposition (truth value).

Some worries

The standard theory is, by and large, incredibly successful.

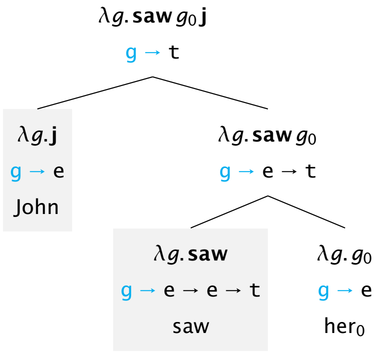
Nevertheless, there are a couple **conceptual issues**:

- *Every* meaning is specified relative to an assignment? Really?
- Binding requires us to gerrymander $[\![\cdot]\!]$ via syncategorematic rules.

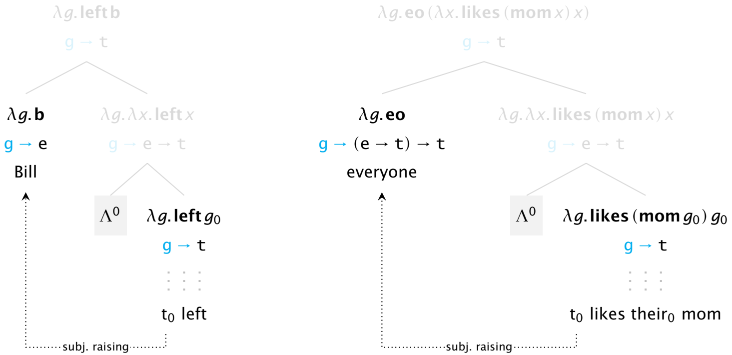
Along with (probably more importantly) a couple **empirical issues**:

- “Paycheck pronouns” are unaccounted for.
- As is binding reconstruction.

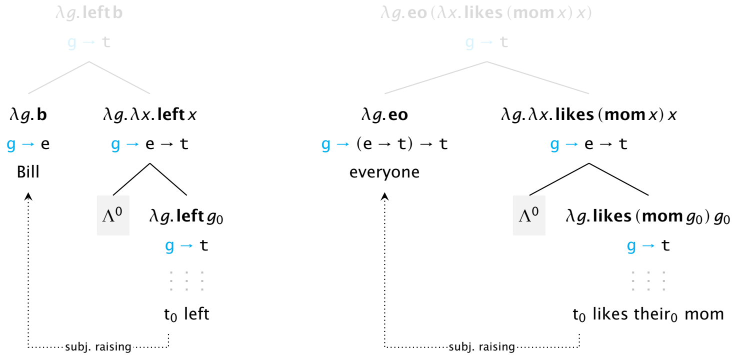
Complicating the lexicon: Nonprominals



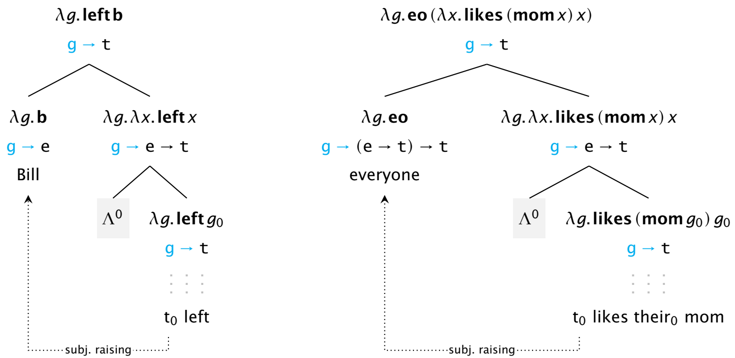
Abstraction goals



Abstraction goals



Abstraction goals



Complicating the grammar: Abstraction

$\lambda g.\, f\, g\, (\mathbf{left}\, g_0)$

$g \to e \to t$

f	$\lambda g.\, \mathbf{left}\, g_0$
$g \to t \to e \to t$	$g \to t$
Λ^0	t_0 left

Complicating the grammar: Abstraction

$\lambda g.\, f\, g\, (\mathbf{left}\, g_0)$

$\mathtt{g} \to \mathtt{e} \to \mathtt{t}$

f

$\mathtt{g} \to \mathtt{t} \to \mathtt{e} \to \mathtt{t}$

Λ^0

$\lambda g.\, \mathbf{left}\, g_0$

$\mathtt{g} \to \mathtt{t}$

t_0 left

No f works in the general case...

Complicating the grammar: Abstraction

$$\lambda g.\, f\, g\, (\mathbf{left}\, g_0)$$

$$g \to e \to t$$

$$f \qquad\qquad \lambda g.\, \mathbf{left}\, g_0$$

$$g \to t \to e \to t \qquad\qquad g \to t$$

$$\Lambda^0 \qquad\qquad t_0 \text{ left}$$

No f works in the general case...

The grammar wants to interpret both branches at *the same assignment*, but the right node must be interpreted at *a shifted assignment*:

$$\underbrace{[\![\Lambda^i\ \alpha]\!]}_{\text{extending } [\![\cdot]\!] \text{ with a syncategorematic rule}} := \lambda g.\lambda x.[\![\alpha]\!]\, g^{i \to x}$$

Under-generation: (binding) reconstruction

It is well known that (quantificational) binding does not require surface c-command (e.g., Sternefeld 1998, 2001, Barker 2012):

1. Which of their$_i$ relatives does everyone$_i$ like _?
2. His$_i$ mom, every boy$_i$ likes _.
3. Their advisor$_i$ seems to every Ph.D. student$_i$ _ to be a genius.
4. Unless he$_i$'s been a bandit, no man$_i$ can be an officer _.

Under-generation: (binding) reconstruction

It is well known that (quantificational) binding does not require surface c-command (e.g., Sternefeld 1998, 2001, Barker 2012):

1. Which of their$_i$ relatives does everyone$_i$ like _?
2. His$_i$ mom, every boy$_i$ likes _.
3. Their advisor$_i$ seems to every Ph.D. student$_i$ _ to be a genius.
4. Unless he$_i$'s been a bandit, no man$_i$ can be an officer _.

But Predicate Abstraction passes modified assignments *down the tree*, and so binding invariably requires (LF) c-command. Scoping the quantifier over the pronoun restores LF c-command, but should trigger a Weak Crossover violation:

5. *Who$_i$ does his$_i$ mother like _?
6. *His$_i$ superior reprimanded no officer$_i$.

Under-generation: Paycheck pronouns

Simple pronouns anaphoric to expressions containing pronouns can receive “sloppy” readings (e.g., Cooper 1979, Engdahl 1986, Jacobson 2000):

1. John$_i$ deposited [his$_i$ paycheck]$_j$, but Bill$_k$ spent it$_j$.
2. Every semanticist$_i$ deposited [their$_i$ paycheck]$_j$. Every philosopher spent it$_j$.

Under-generation: Paycheck pronouns

Simple pronouns anaphoric to expressions containing pronouns can receive “sloppy” readings (e.g., Cooper 1979, Engdahl 1986, Jacobson 2000):

1. John$_i$ deposited [his$_i$ paycheck]$_j$, but Bill$_k$ spent it$_j$.
2. Every semanticist$_i$ deposited [their$_i$ paycheck]$_j$. Every philosopher spent it$_j$.

These are unaccounted for on the standard picture. There are two (related) issues:

- What’s the denotation of the paycheck pronoun? Can’t be a simple individual.
- The paycheck pronoun’s meaning is different from the thing it’s anaphoric to.

Roadmap

The theoretical baggage associated with the standard account is straightforward and cheap to dispense with, via something called an **applicative functor**.

The empirical baggage seems to require an additional piece for dealing with *higher-order meanings*. This upgrades the applicative functor into a **monad**.

Getting modular

Abstraction and modularity

Functional programmers are lazy. When they see a bit of code occurring over and over again in their program, they abstract it out as a separate function.

Let's see how far this gets us.

For assignment-dependence

The idea — almost embarrassing in its simplicity — is to just abstract out and modularize the core features of the standard account.

For assignment-dependence

The idea — almost embarrassing in its simplicity — is to just abstract out and modularize the core features of the standard account.

In lieu of treating everything as trivially dependent on an assignment, invoke a function ρ which turns any x into a constant function from assignments into x:

$$\rho := \underbrace{\lambda x.\lambda g.x}_{a \to \mathsf{g} \to a}$$

For assignment-dependence

The idea — almost embarrassing in its simplicity — is to just abstract out and modularize the core features of the standard account.

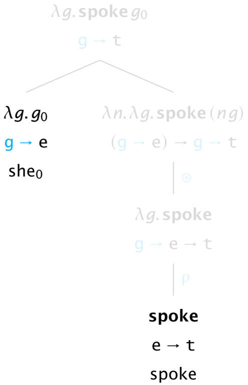
In lieu of treating everything as trivially dependent on an assignment, invoke a function ρ which turns any x into a constant function from assignments into x:

$$\rho := \underbrace{\lambda x.\lambda g.x}_{a \to \mathsf{g} \to a}$$

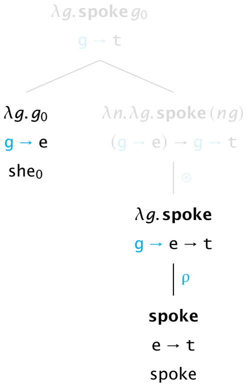
Instead of taking on $[\![\cdot]\!]$ wholesale, we'll help ourselves to a function $\circledast$ which allows us to perform assignment-friendly function application on demand:

$$\circledast := \underbrace{\lambda m.\lambda n.\lambda g.\, mg\,(ng)}_{(\mathsf{g} \to a \to b) \to (\mathsf{g} \to a) \to \mathsf{g} \to b}$$

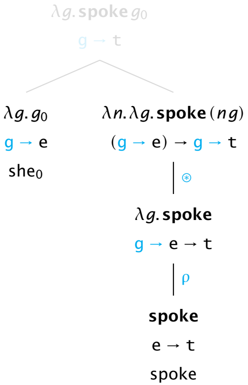
Sample derivations



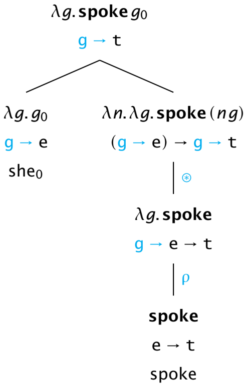
Sample derivations



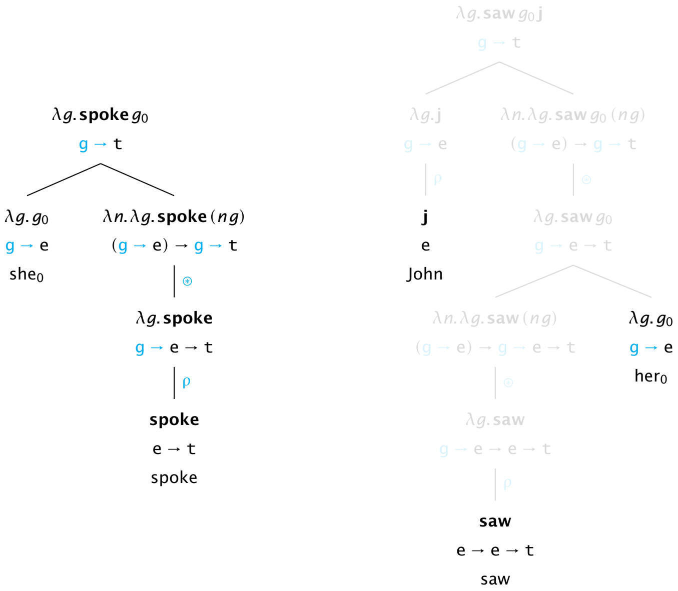
Sample derivations



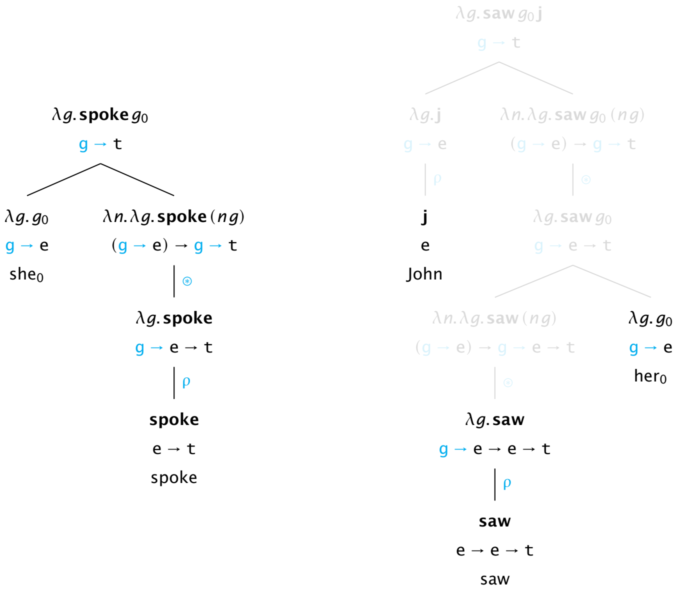
Sample derivations



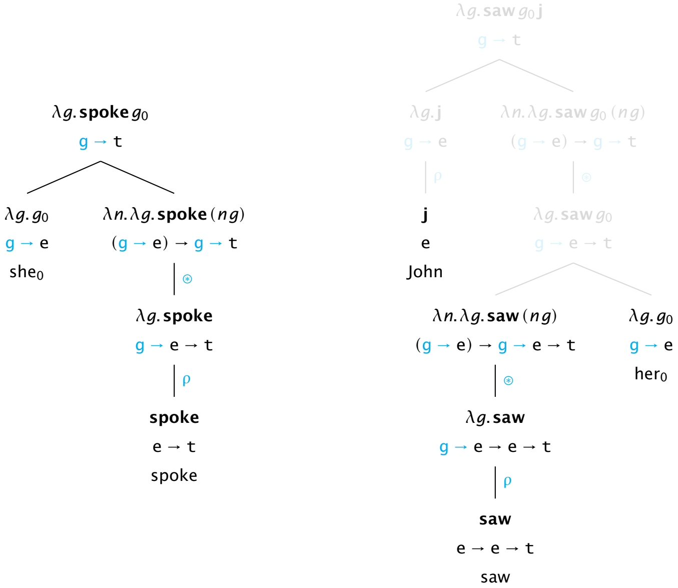
Sample derivations



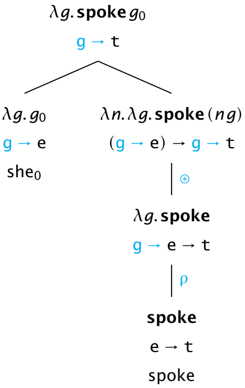
Sample derivations

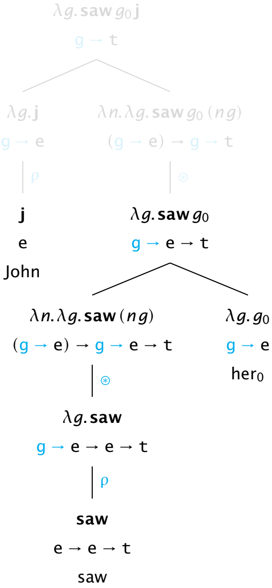


Sample derivations

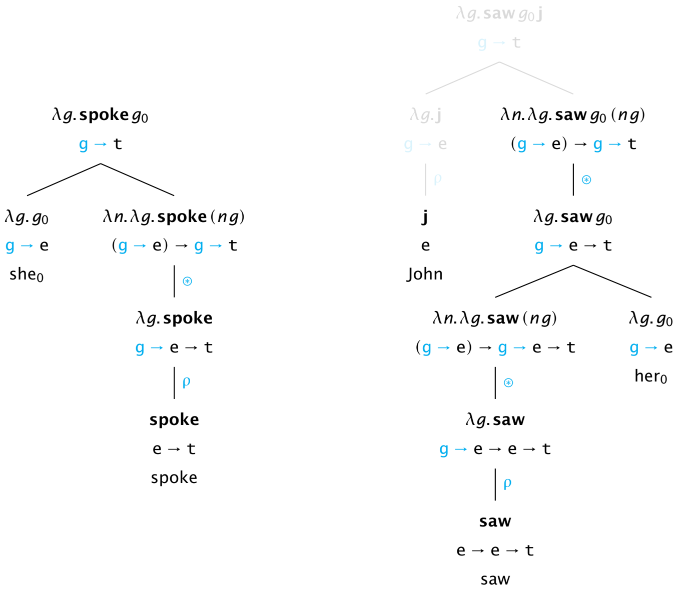


Sample derivations

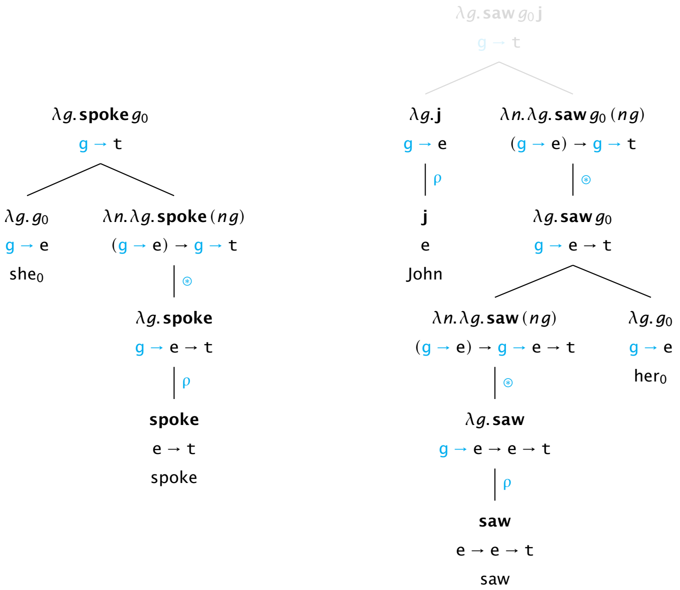




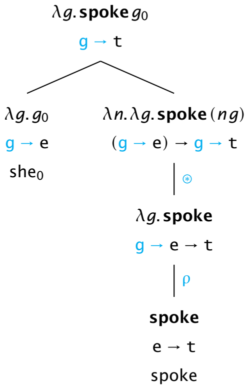
Sample derivations

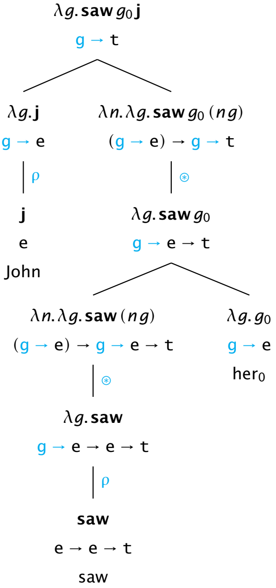


Sample derivations



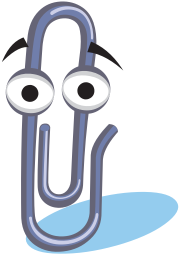
Sample derivations





Basically

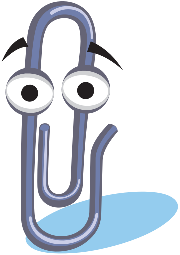
Basically



It looks like you’re trying to do semantics.

Would you like help?

Basically

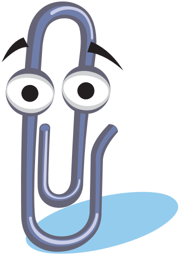


It looks like you're trying to do semantics.

Would you like help?

- ☐ Give me a ρ

Basically

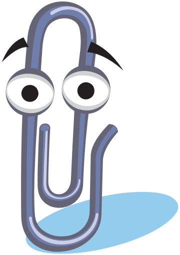


It looks like you're trying to do semantics.

Would you like help?

- ☐ Give me a ρ
- ☐ Give me a $\circledast$

Basically

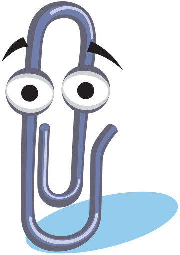


It looks like you're trying to do semantics.

Would you like help?

- ☐ Give me a ρ
- ☐ Give me a $\circledast$
- ☐ Don't show me this tip again

Basically



It looks like you're trying to do semantics.

Would you like help?

- ☐ Give me a ρ
- ☑ Give me a $\circledast$
- ☐ Don't show me this tip again

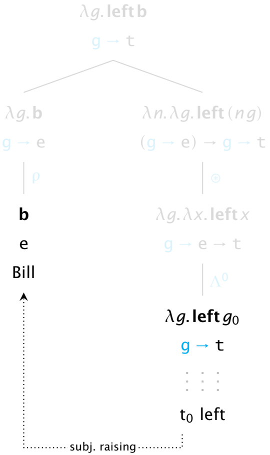
Conceptual issues dissolved

First, ρ allows stuff that's not really assignment-dependent to be lexically so.

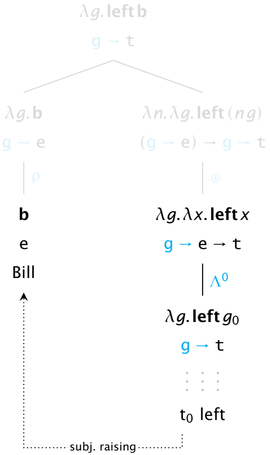
Second, because the grammar doesn't *insist on* composing meanings with $[\![\cdot]\!]$, abstraction can be defined directly (e.g., Sternefeld 1998, 2001, Kobele 2010):

$$\Lambda^i := \underbrace{\lambda f.\lambda g.\lambda x.f\,g^{i\to x}}_{(\mathsf{g}\to a)\to\mathsf{g}\to b\to a}$$

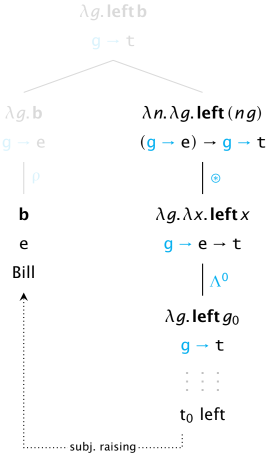
$$\Lambda^i := \lambda f.\lambda g.\lambda x.f\,g^{i\to x}$$



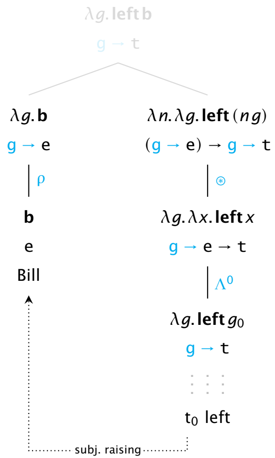
$$\Lambda^i := \lambda f.\,\lambda g.\,\lambda x.\, f\, g^{i\to x}$$



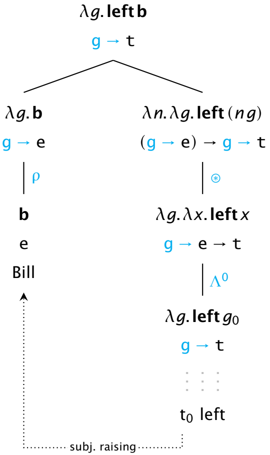
$$\Lambda^i := \lambda f.\lambda g.\lambda x.f\, g^{i\to x}$$



$$\Lambda^i := \lambda f.\lambda g.\lambda x.f\, g^{i\to x}$$



$$\Lambda^i := \lambda f.\lambda g.\lambda x.f\, g^{i\to x}$$



A familiar construct

When we abstract out ρ and $\circledast$ in this way, we're in the presence of something known to computer scientists and functional programmers as an **applicative functor** (McBride & Paterson 2008, Kiselyov 2015).

You might also recognize ρ and $\circledast$ as the $\mathbb{K}$ and $\mathbb{S}$ combinators from Combinatory Logic (Curry & Feys 1958).

Applicative functors

An applicative functor is a type constructor F with two functions:

$$\rho :: a \to F\,a \qquad \circledast :: F\,(a \to b) \to F\,a \to F\,b$$

Satisfying a few laws:

Homomorphism

$$\rho\, f \circledast \rho\, x = \rho\,(f\,x)$$

Identity

$$\rho\,(\lambda x.x) \circledast v = v$$

Interchange

$$\rho\,(\lambda f.f\,x) \circledast u = u \circledast \rho\, x$$

Composition

$$\rho\,(\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$$

Basically, these laws say that $\circledast$ should be a kind of fancy functional application, and ρ should be a trivial way to make something fancy.

Generality

Another example of an applicative functor, for sets:

$$\rho\, x := \{x\} \qquad m \circledast n := \{f\,x \mid f \in m, x \in n\}$$

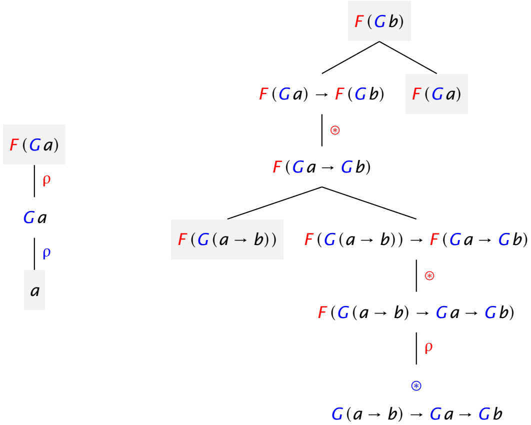
(See Charlow 2014, 2017 for more on this.)

The technique is super general, and can be fruitfully applied (inter alia) to dynamics, presupposition, supplementation, (association with) focus, and scope:

$$\rho\, x := \lambda k.\, k\, x \qquad m \circledast n := \lambda k.\, m\,(\lambda f.\, n\,(\lambda x.\, k\,(f\,x)))$$

(See Shan & Barker 2006, Barker & Shan 2008 for more on this.)

Applicative functors compose



Whenever you have two applicative functors, you’re *guaranteed* to have two more!

Getting higher-order

What we have

The applicative-functor approach to assignment sensitivity immediately dissolves the theoretical baggage associated with the standard account:

- ρ allows us to keep the lexicon maximally simple.
- $\circledast$ liberates us from $[\![\cdot]\!]$, allowing a categorematic treatment of abstraction.

All in all, this seems like a nice grammatical interface for pronouns and binding. Extra resources are invoked only when they're required for composition.

What we don't have

However, it seems ρ and $\circledast$ are no help for reconstruction or paychecks (time permitting, I'll question this point at the end, but let's run with it for now).

Intuitively, both phenomena are *higher-order*: the referent anaphorically retrieved by the paycheck pronoun or the topicalized expression's trace is an 'intension', rather than an 'extension' (cf. Sternefeld 1998, 2001, Hardt 1999, Kennedy 2014).

1. John$_i$ deposited [his$_i$ paycheck]$_j$, but Bill$_k$ spent it$_j$.
2. [His$_i$ mom]$_j$, every boy$_i$ likes t$_j$.

Anaphora to intensions

What would it mean for a pronoun (or trace) to be anaphoric to an intension?

Anaphora to intensions

What would it mean for a pronoun (or trace) to be anaphoric to an intension?

Perhaps: the value returned at an assignment (the anaphorically retrieved meaning) is still sensitive to an assignment (i.e., intensional).

$$g \to g \to e$$

Anaphora to intensions

What would it mean for a pronoun (or trace) to be anaphoric to an intension?

Perhaps: the value returned at an assignment (the anaphorically retrieved meaning) is still sensitive to an assignment (i.e., intensional).

$$\mathsf{g} \to \mathsf{g} \to \mathsf{e}$$

Going whole hog, pronouns have a generalized, recursive type:

$$\mathfrak{pro} ::= \mathsf{g} \to \mathsf{e} \mid \underbrace{\mathsf{g} \to \mathfrak{pro}}_{\mathsf{g}\to\mathsf{g}\to\mathsf{e},\ \mathsf{g}\to\mathsf{g}\to\mathsf{g}\to\mathsf{e},\ \ldots}$$

But, importantly, a unitary lexical semantics: $[\![\text{she}_0]\!] := \lambda g.\, g_0$.

μ for higher-order pronouns

Higher-order pronoun meanings require a higher-order combinator:

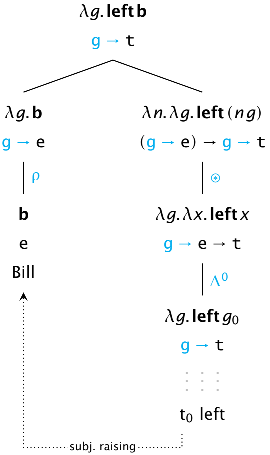
$$\mu := \underbrace{\lambda m. \lambda g. m\,g\,g}_{(\mathsf{g} \to \mathsf{g} \to a) \to \mathsf{g} \to a}$$

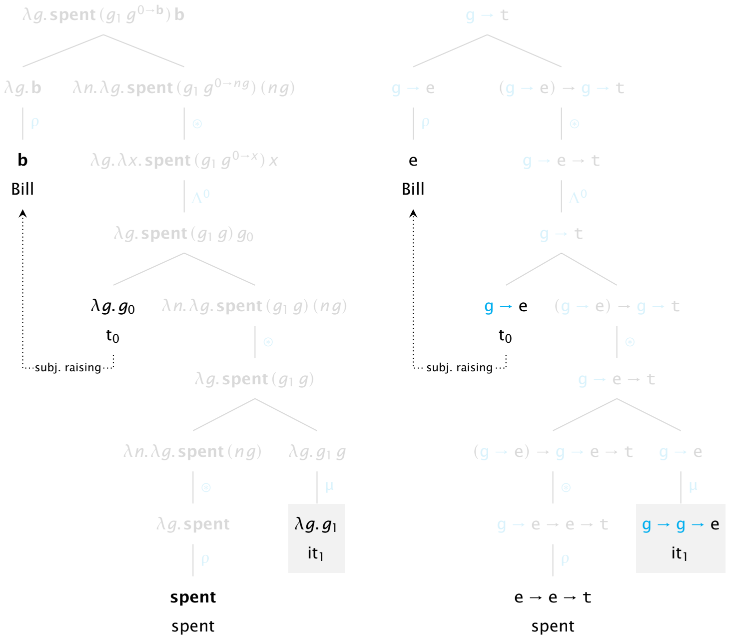
(Aka the $\mathbb{W}$ combinator from Combinatory Logic.)

μ takes an expression *m* that's anaphoric to an intension, and obtains an extension by evaluating the anaphorically retrieved intension $m\,g$ once more against g. In other words, it turns a higher-order pronoun meaning into a garden-variety one:

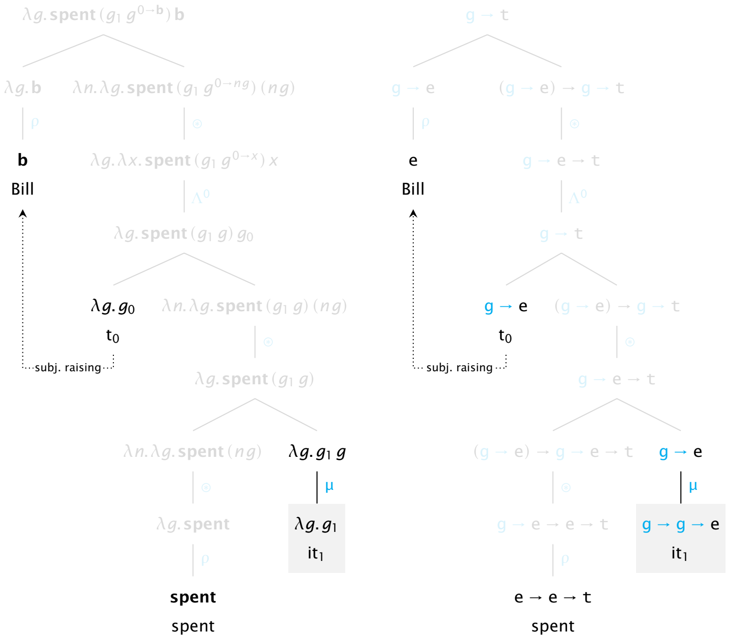
$$\mu\,(\underbrace{\lambda g. g_0}_{\mathsf{g} \to \mathsf{g} \to \mathsf{e}}) = \underbrace{\lambda g. g_0\,g}_{\mathsf{g} \to \mathsf{e}}$$

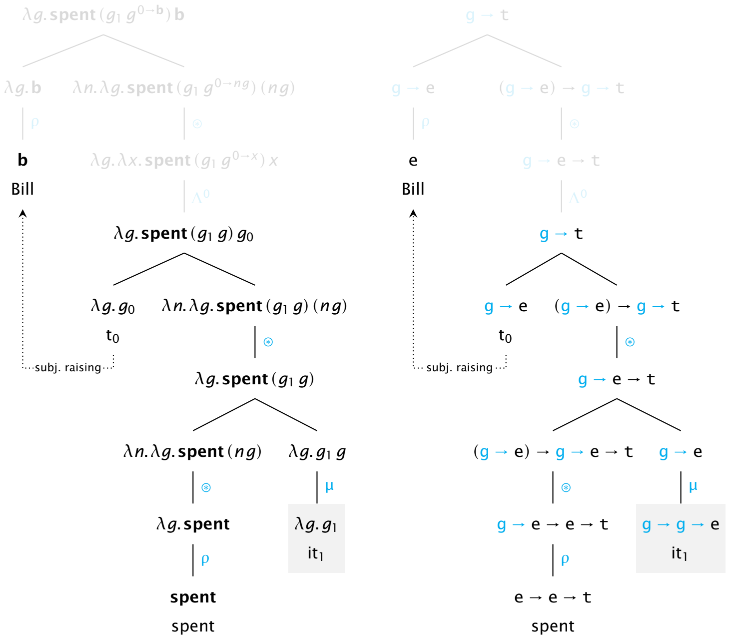
Recalling our binding derivation

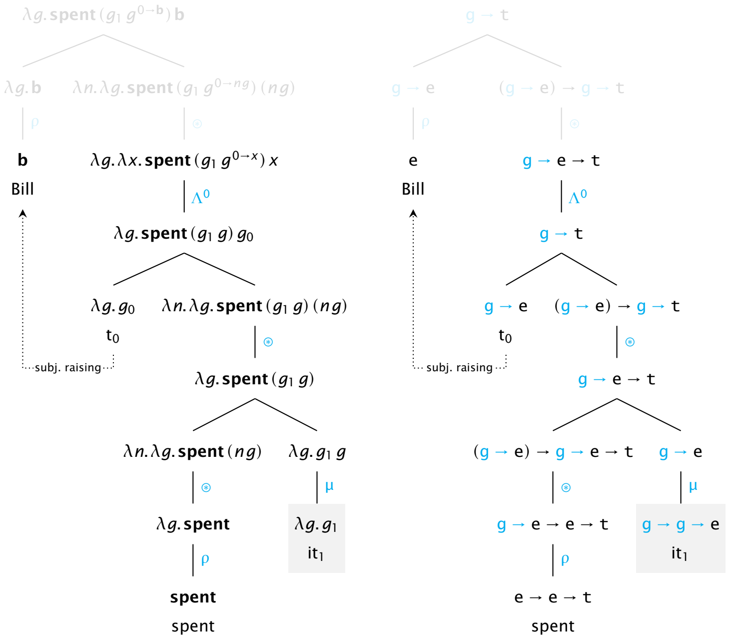


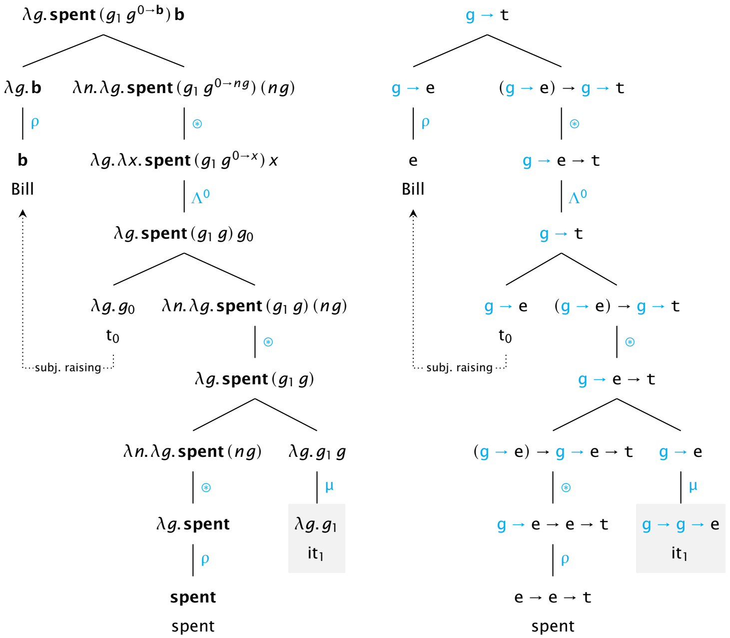


λg. spent (g1 g0→b) b
λg. b
ρ
λn.λg. spent (g1 g0→ng) (ng)
⊛
b
Bill
λg.λx. spent (g1 g0→x) x
Λ0
λg. spent (g1 g) g0
λg. g0
t0
λn.λg. spent (g1 g) (ng)
subj. raising
λg. spent (g1 g)
λn.λg. spent (ng)
λg. g1 g
μ
λg. spent
λg. g1
it1
spent
spent
g → t
g → e
(g → e) → g → t
e
Bill
g → e → t
g → t
g → e
t0
(g → e) → g → t
subj. raising
g → e → t
(g → e) → g → e → t
g → e
g → e → e → t
g → g → e
it1
e → e → t
spent


λg. spent (g1 g0→b) b
λg. b
λn.λg. spent (g1 g0→ng) (ng)
ρ
⊛
b
Bill
λg.λx. spent (g1 g0→x) x
Λ0
λg. spent (g1 g) g0
λg. g0
t0
λn.λg. spent (g1 g) (ng)
subj. raising
λg. spent (g1 g)
λn.λg. spent (ng)
λg. g1 g
μ
λg. spent
λg. g1
it1
spent
spent
g → t
g → e
(g → e) → g → t
e
Bill
g → e → t
g → t
g → e
t0
(g → e) → g → t
subj. raising
g → e → t
(g → e) → g → e → t
g → e
μ
g → e → e → t
g → g → e
it1
e → e → t
spent


λg. spent (g₁ g^{0→b}) b
λg. b
λn.λg. spent (g₁ g^{0→ng}) (ng)
ρ
⊛
b
Bill
λg.λx. spent (g₁ g^{0→x}) x
Λ⁰
λg. spent (g₁ g) g₀
λg. g₀
t₀
λn.λg. spent (g₁ g) (ng)
⊛
subj. raising
λg. spent (g₁ g)
λn.λg. spent (ng)
λg. g₁ g
⊛
μ
λg. spent
λg. g₁
it₁
ρ
spent
spent
g → t
g → e
(g → e) → g → t
ρ
⊛
e
Bill
g → e → t
Λ⁰
g → t
g → e
(g → e) → g → t
t₀
⊛
subj. raising
g → e → t
(g → e) → g → e → t
g → e
⊛
μ
g → e → e → t
g → g → e
it₁
ρ
e → e → t
spent


λg. spent (g1 g0→b) b
λg. b
λn.λg. spent (g1 g0→ng) (ng)
b
Bill
λg.λx. spent (g1 g0→x) x
Λ0
λg. spent (g1 g) g0
λg. g0
t0
λn.λg. spent (g1 g) (ng)
subj. raising
λg. spent (g1 g)
λn.λg. spent (ng)
λg. g1 g
μ
λg. spent
λg. g1
it1
ρ
spent
spent
g → t
g → e
(g → e) → g → t
e
Bill
g → e → t
Λ0
g → t
g → e
t0
(g → e) → g → t
subj. raising
g → e → t
(g → e) → g → e → t
g → e
μ
g → e → e → t
g → g → e
it1
ρ
e → e → t
spent


λg. spent (g₁ g^{0→b}) b
λg. b
λn.λg. spent (g₁ g^{0→ng}) (ng)
ρ
⊛
b
Bill
λg.λx. spent (g₁ g^{0→x}) x
Λ⁰
λg. spent (g₁ g) g₀
λg. g₀
t₀
λn.λg. spent (g₁ g) (ng)
subj. raising
λg. spent (g₁ g)
λn.λg. spent (ng)
λg. g₁ g
μ
λg. spent
λg. g₁
it₁
spent
spent
g → t
g → e
(g → e) → g → t
e
Bill
g → e → t
Λ⁰
g → t
g → e
t₀
(g → e) → g → t
subj. raising
g → e → t
(g → e) → g → e → t
g → e
g → e → e → t
g → g → e
it₁
e → e → t
spent

Taking stock

The derived meaning is $\lambda g.\,\mathbf{spent}\,(g_1\ g^{0\to\mathbf{b}})\,\mathbf{b}$. If the incoming assignment assigns 1 to $\lambda g.\,\mathbf{paycheck}\,g_0$ (the intension of *his*$_0$ *paycheck*), we're home free.

Aside from the type assigned to *her*$_1$ and the invocation of μ, this derivation is exactly the same as a normal case of pronominal binding.

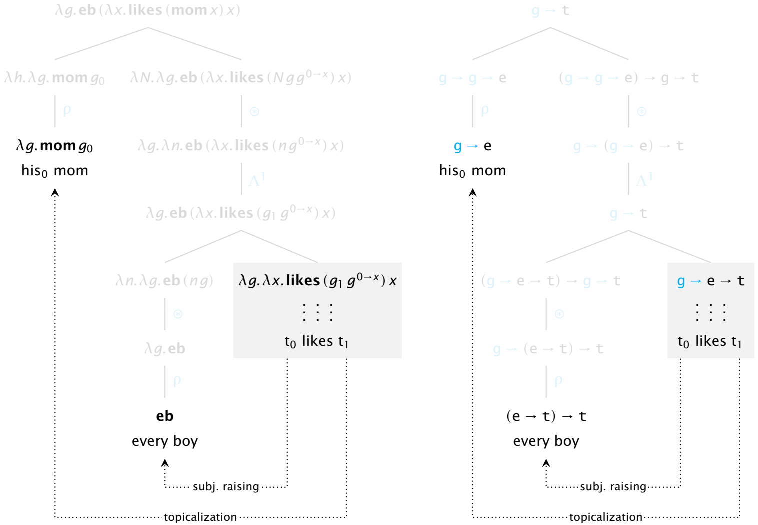
The secret sauce is generalizing the types of pronouns (but not their lexical semantics!), and invoking μ for higher-typed pronoun instantiations.

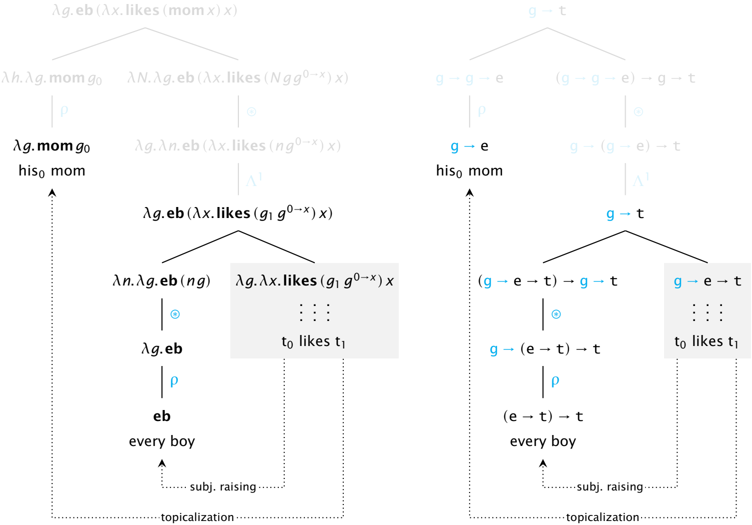
Reconstruction works the same

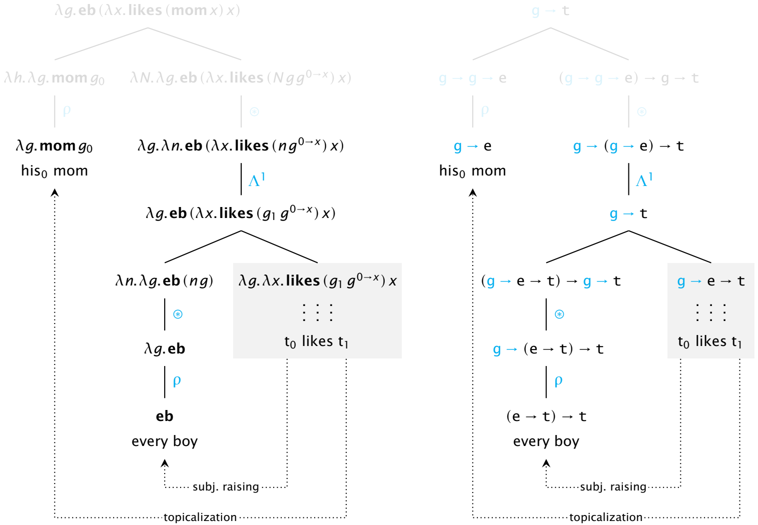
We can pull off a similar trick for reconstruction: treat the trace as higher-order, making it anaphoric to the *intension* of the topicalized expression.

1. [His$_i$ mom]$_j$, every boy$_i$ likes t$_j$.

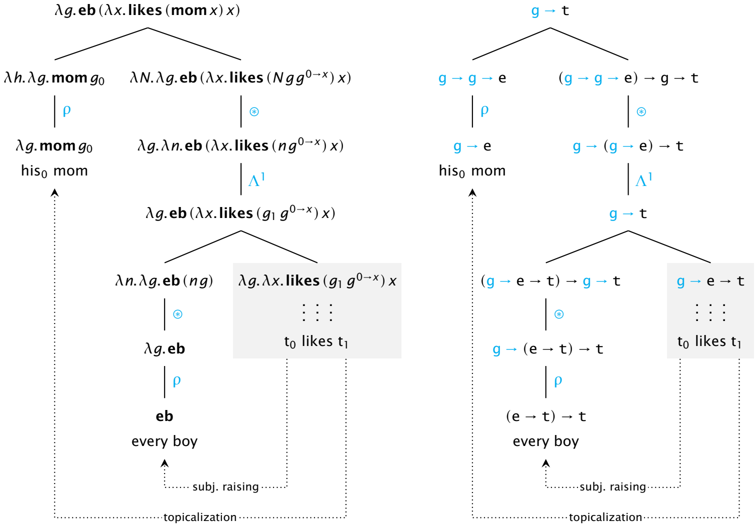
Use μ to make sure everything fits together, and we're done.


λg. eb (λx. likes (mom x) x)
λh.λg. mom g0
λN.λg. eb (λx. likes (N g g0→x) x)
ρ
⊛
λg. mom g0
his0 mom
λg.λn. eb (λx. likes (n g0→x) x)
Λ1
λg. eb (λx. likes (g1 g0→x) x)
λn.λg. eb (n g)
λg.λx. likes (g1 g0→x) x
t0 likes t1
λg. eb
eb
every boy
subj. raising
topicalization
g → t
g → g → e
(g → g → e) → g → t
g → e
his0 mom
g → (g → e) → t
g → t
(g → e → t) → g → t
g → e → t
t0 likes t1
g → (e → t) → t
(e → t) → t
every boy
subj. raising
topicalization


λg. eb (λx. likes (mom x) x)
λh.λg. mom g0
λN.λg. eb (λx. likes (N g g0→x) x)
ρ
⊛
λg. mom g0
his0 mom
λg.λn. eb (λx. likes (n g0→x) x)
Λ1
λg. eb (λx. likes (g1 g0→x) x)
λn.λg. eb (n g)
λg.λx. likes (g1 g0→x) x
⊛
λg. eb
t0 likes t1
ρ
eb
every boy
subj. raising
topicalization
g → t
g → g → e
(g → g → e) → g → t
ρ
⊛
g → e
his0 mom
g → (g → e) → t
Λ1
g → t
(g → e → t) → g → t
g → e → t
⊛
g → (e → t) → t
t0 likes t1
ρ
(e → t) → t
every boy
subj. raising
topicalization



λg. eb (λx. likes (mom x) x)
λh.λg. mom g0
λN.λg. eb (λx. likes (N g g0→x) x)
ρ
⊛
λg. mom g0
his0 mom
λg.λn. eb (λx. likes (n g0→x) x)
Λ1
λg. eb (λx. likes (g1 g0→x) x)
λn.λg. eb (n g)
λg.λx. likes (g1 g0→x) x
λg. eb
t0 likes t1
eb
every boy
subj. raising
topicalization
g → t
g → g → e
(g → g → e) → g → t
g → e
g → (g → e) → t
g → e → t
(g → e → t) → g → t
g → (e → t) → t
(e → t) → t


λg. eb (λx. likes (mom x) x)
λh.λg. mom g0
λN.λg. eb (λx. likes (N g g0→x) x)
ρ
⊛
λg. mom g0
his0 mom
λg.λn. eb (λx. likes (n g0→x) x)
Λ1
λg. eb (λx. likes (g1 g0→x) x)
λn.λg. eb (n g)
λg.λx. likes (g1 g0→x) x
λg. eb
t0 likes t1
eb
every boy
subj. raising
topicalization
g → t
g → g → e
(g → g → e) → g → t
g → e
g → (g → e) → t
g → t
(g → e → t) → g → t
g → e → t
g → (e → t) → t
(e → t) → t
every boy
subj. raising
topicalization

Another familiar construct

Our grammatical interface for pronouns and binding has three pieces: ρ, $\circledast$, and μ.

ρ and $\circledast$ form an applicative functor. Does this suite of three combinators also correspond to something interesting?

Another familiar construct

Our grammatical interface for pronouns and binding has three pieces: ρ, $\circledast$, and μ.

ρ and $\circledast$ form an applicative functor. Does this suite of three combinators also correspond to something interesting? Yes, it's a **monad** (Moggi 1989, Wadler 1992, 1995, Shan 2002, Giorgolo & Asudeh 2012, Charlow 2014, 2017, . . .).

Equivalence of definitions

The usual presentation of monads is in terms of two functions η and ≫=:

$$\eta \;::\; a \to T\,a$$
$$\gg\!= \;::\; T\,a \to (a \to T\,b) \to T\,b$$

(Satisfying certain laws, just like applicative functors.)

For the present case, the monad of interest is known as the Environment or Reader monad. Its η is just the same as ρ. Its ≫= is:

$$\gg\!= \;:=\; \underbrace{\lambda m.\lambda f.\lambda g.f\,(m\,g)\,g}_{(\mathrm{g}\to \mathrm{a})\to(a\to \mathrm{g}\to b)\to \mathrm{g}\to b}$$

Monads don't compose

There is an extremely sad fact about monads: unlike applicatives, they do not freely compose! If you have two monads, there is no guarantee you will have a third, and no general recipe for composing monads to yield new ones.

So applicatives are easy to work with in isolation. You can be confident that they will play nicely with other applicative things in your grammar. Monads, not so much.

Monads don't compose

There is an extremely sad fact about monads: unlike applicatives, they do not freely compose! If you have two monads, there is no guarantee you will have a third, and no general recipe for composing monads to yield new ones.

So applicatives are easy to work with in isolation. You can be confident that they will play nicely with other applicative things in your grammar. Monads, not so much.

> *The moral is this: if you have got an Applicative functor, that is good; if you have also got a Monad, that is even better!*

Monads don't compose

There is an extremely sad fact about monads: unlike applicatives, they do not freely compose! If you have two monads, there is no guarantee you will have a third, and no general recipe for composing monads to yield new ones.

So applicatives are easy to work with in isolation. You can be confident that they will play nicely with other applicative things in your grammar. Monads, not so much.

> *The moral is this: if you have got an Applicative functor, that is good; if you have also got a Monad, that is even better! And the dual of the moral is this: if you need a Monad, that is fine; if you need only an Applicative functor, that is even better!*
>
> (McBride & Paterson 2008: 8)

Variable-free semantics

Pronouns as identity maps

Jacobson proposes we stop thinking of pronouns as assignment-relative and index-oriented. Instead, she suggests we model pronouns as **identity functions**:

$$\textbf{she} := \underbrace{\lambda x.\, x}_{e \to e}$$

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

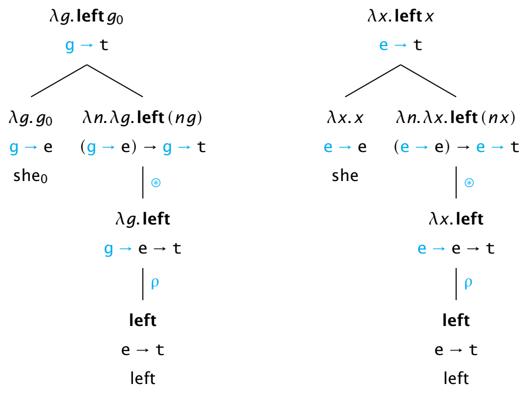
Pronouns as identity maps

Jacobson proposes we stop thinking of pronouns as assignment-relative and index-oriented. Instead, she suggests we model pronouns as **identity functions**:

$$\textbf{she} := \underbrace{\lambda x.\, x}_{e \to e}$$

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

Of course, this is *exactly* the same problem that comes up when you introduce assignment-dependent meanings! And hence it admits the exact same solution.



In an important sense, then, the compositional apparatus underwriting variable-free composition is equivalent to that underwriting assignment-friendly composition!

Heim & Kratzer (1998: 92): $[\![t]\!] = \lambda x.x$

(5) Preliminary definition: An *assignment* is an individual (that is, an element of D (= D_e)).

A trace under a given assignment denotes the individual that constitutes that assignment; for example:

(6) The denotation of "t" under the assignment Texas is Texas.

An appropriate notation to abbreviate such statements needs to be a little more elaborate than the simple $[\![\ldots]\!]$ brackets we have used up to now. We will indicate the assignment as a superscript on the brackets; for instance, (7) will abbreviate (6):

(7) $[\![t]\!]^{\text{Texas}}$ = Texas.

The general convention for reading this notation is as follows: Read "$[\![\alpha]\!]^a$" as "the denotation of α under a" (where α is a tree and a is an assignment).

(7) exemplifies a special case of a general rule for the interpretation of traces, which we can formulate as follows:

(8) If α is a trace, then, for any assignment a, $[\![\alpha]\!]^a = a$.

Small caveat

This is not exactly how Jacobson's system works. For one, it's overlaid on a categorial syntax. For another, Jacobson's key combinator is **Z**:

$$\mathbf{Z} := \lambda f.\lambda m.\lambda x.f\,(m\,x)\,x$$

Does this remind you of anything?

Small caveat

This is not exactly how Jacobson's system works. For one, it's overlaid on a categorial syntax. For another, Jacobson's key combinator is **Z**:

$$\mathbf{Z} := \lambda f.\lambda m.\lambda x.f\,(m\,x)\,x$$

Does this remind you of anything?

It's (a slightly shuffled version of) the $\gg\!=$ operation from the Reader monad!

$$\gg\!= \;:= \lambda m.\lambda f.\lambda g.f\,(m\,g)\,g$$

Multiple pronouns

There is an important difference between assignments and individuals as reference-fixing devices. Assignments are data structures that can in principle value *every* free pronoun you need. But an individual can only value *co-valued* pronouns!

1. She saw her.

So a variable-free treatment of cases like these will inevitably give you something like the following (I will spare you the details):

$$\lambda x.\lambda y.\mathbf{saw}\, y\, x$$

Assignments, and "variables", on demand

$$\frac{|(a \times b) \to c|}{|a \to b \to c|} = \frac{|c|^{|a| \cdot |b|}}{(|c|^{|b|})^{|a|}} = 1$$

Witness the curry/uncurry isomorphisms:

$$\mathbf{curry}\, f := \lambda x.\lambda y.f\,(x,y) \qquad \mathbf{uncurry}\, f := \lambda(x,y).f\,x\,y$$

In other words, by (iteratively) uncurrying a variable-free proposition, you end up with a dependence on a sequence (tuple) of things. Essentially, an assignment.

$$\mathbf{uncurry}\,(\lambda x.\lambda y.\mathbf{saw}\,y\,x) = \lambda(x,y).\mathbf{saw}\,y\,x = \lambda p.\mathbf{saw}\,p_1\,p_0$$

Obversely, by iteratively currying a sequence(/tuple)-dependent proposition, you end up with a higher-order function. Essentially, a variable-free meaning.

$$\mathbf{curry}\,(\lambda p.\mathbf{saw}\,p_1\,p_0) = \mathbf{curry}\,(\lambda(x,y).\mathbf{saw}\,y\,x) = \lambda x.\lambda y.\mathbf{saw}\,y\,x$$

Variable-free semantics?

So variable-free semantics (can) have the same combinatorics as the variable-full semantics. This is no great surprise: they're both about compositionally dealing with "incomplete" meanings.

Moreover, under the curry/uncurry isomorphisms, a variable-free proposition is equivalent to (something) like an assignment dependent proposition.

Let's call the whole thing off?

Back to applicatives

A bit o' type theory

What is the *type* of an assignment function?

A bit o' type theory

What is the *type* of an assignment function? Standardly, $g ::= \mathbb{N} \to e$.

A bit o' type theory

What is the *type* of an assignment function? Standardly, $g ::= \mathbb{N} \to e$. But we want assignment functions to harbor values of *all sorts of types*, for binding reconstruction and paychecks, cross-categorial topicalization, scope reconstruction.

A bit o' type theory

What is the *type* of an assignment function? Standardly, $g ::= \mathbb{N} \to e$. But we want assignment functions to harbor values of *all sorts of types*, for binding reconstruction and paychecks, cross-categorial topicalization, scope reconstruction.

Muskens (1995) cautions against trying to pack too much into our assignments:

$$\textbf{[AX1]} \quad \forall g, n, x_\alpha : \exists h : g[n]\, h \wedge h_n = x \quad \text{for all } \alpha \in \Theta$$

A bit o' type theory

What is the *type* of an assignment function? Standardly, $g ::= \mathbb{N} \rightarrow e$. But we want assignment functions to harbor values of *all sorts of types*, for binding reconstruction and paychecks, cross-categorial topicalization, scope reconstruction.

Muskens (1995) cautions against trying to pack too much into our assignments:

$$[\textbf{AX1}] \quad \forall g, n, x_\alpha : \exists h : g[n]\, h \wedge h_n = x \quad \text{for all } \alpha \in \Theta$$

If, say, $g \rightarrow e \in \Theta$, this ends up paradoxical! **AX1** requires there to be as many assignments as there are functions from assignments to individuals: $|g| \geqslant |g \rightarrow e|$.

A hierarchy of assignments?

We might try parametrizing assignments by the types of things they harbor:

$$g_a ::= \mathbb{N} \to a$$

(An *a*-assignment is a function from indices into inhabitants of *a*.)

This is no longer paradoxical: we have a hierarchy of assignments, much like we have a hierarchy of types.

This is *weird*

If $g_a ::= \mathbb{N} \to a$, what type is the blue part in the following?

1. ...and $[_{VP}$ buy the couch$]_1$ she_0 did $[_{VP}\ t_1]$.

A couple of countervailing considerations:

- There's a (free) pronoun, $\therefore\ g_e \to t$?
- There's a (free) VP variable, $\therefore\ g_{e \to t} \to t$?

Splitting the difference: $g_{e \to t} \to g_e \to t$.

A non-problem

So is there a univocal type for propositions? If so, is every lexical entry specified relative to an infinite sequence of assignments, one per type in the inductive type hierarchy?

A non-problem

So is there a univocal type for propositions? If so, is every lexical entry specified relative to an infinite sequence of assignments, one per type in the inductive type hierarchy? That seems bad.

A non-problem

So is there a univocal type for propositions? If so, is every lexical entry specified relative to an infinite sequence of assignments, one per type in the inductive type hierarchy? That seems bad.

But it is only bad if you're working in the old, one-size-fits-all paradigm.

A non-problem

So is there a univocal type for propositions? If so, is every lexical entry specified relative to an infinite sequence of assignments, one per type in the inductive type hierarchy? That seems bad.

But it is only bad if you're working in the old, one-size-fits-all paradigm.

Back to intensional pronouns (and traces)

With type-segregated assignment, we'll have the following for a paycheck pronoun or reconstruction-ready trace:

$$\lambda g.\, g_0 \equiv_\lambda \underbrace{\lambda g.\, \lambda h.\, g_0\, h}_{\mathtt{g}_{\mathtt{g}_{\mathtt{e}\to\mathtt{e}}} \to \mathtt{g}_{\mathtt{e}} \to \mathtt{e}}$$

This projects up into the following meaning for a paycheck sentence (composition here involves composing our old applicative with *itself*):

$$\underbrace{\lambda g.\, \lambda h.\, \textbf{spent}\,(g_1\, h^{0\to\mathbf{b}})\, \mathbf{b}}_{\mathtt{g}_{\mathtt{g}_{\mathtt{e}\to\mathtt{e}}} \to \mathtt{g}_{\mathtt{e}} \to \mathtt{t}}$$

So our sentence depends on two assignments.

An applicative after all

The pressure to μ is the pressure to assign a uniform type to sentences. Things that depend on two assignments need to be turned into things that depend on just one, so that our sentence can depend on just one.

There are reasons to want that if you're working in the standard mold. There are no reasons to want that if your perspective on composition is more modular. And there are reasons to *disprefer* that if you're going variable-free.

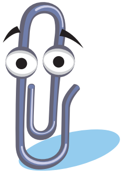
Remember the dual of the moral: if you need a Monad, that is fine; if you need only an Applicative functor, that is even better!

Concluding

Getting modular (either via applicative functors or monads) dissolves theoretical and empirical issues characteristic of one-size-fits-all approaches to composition.

Once we take a modular view on assignment-dependence, a strong parallel between variable-free and variable-full approaches comes into view.

Don't tie your hands if you don't have to.



Barker, Chris. 2012. Quantificational binding does not require c-command. *Linguistic Inquiry* 43(4). 614–633. https://doi.org/doi:10.1162/ling_a_00108.

Barker, Chris & Chung-chieh Shan. 2008. Donkey anaphora is in-scope binding. *Semantics and Pragmatics* 1(1). 1–46. https://doi.org/10.3765/sp.1.1.

Charlow, Simon. 2014. *On the semantics of exceptional scope*. New York University Ph.D. thesis. http://semanticsarchive.net/Archive/2JmMWRjY/.

Charlow, Simon. 2017. The scope of alternatives: Indefiniteness and islands. Unpublished ms. http://ling.auf.net/lingbuzz/003302.

Cooper, Robin. 1979. The interpretation of pronouns. In Frank Heny & Helmut S. Schnelle (eds.), *Syntax and semantics, volume 10: Selections from the Third Groningen Round Table*, 61–92. New York: Academic Press.

Curry, Haskell B. & Robert Feys. 1958. *Combinatory logic*. Vol. 1. Amsterdam: North Holland.

Engdahl, Elisabet. 1986. *Constituent questions*. Vol. 27 (Studies in Linguistics and Philosophy). Dordrecht: Reidel. https://doi.org/10.1007/978-94-009-5323-9.

Giorgolo, Gianluca & Ash Asudeh. 2012. $(M, \eta, \star)$: Monads for conventional implicatures. In Ana Aguilar Guevara, Anna Chernilovskaya & Rick Nouwen (eds.), *Proceedings of Sinn und Bedeutung 16*, 265–278. MIT Working Papers in Linguistics. http://mitwpl.mit.edu/open/sub16/Giorgolo.pdf.

Hardt, Daniel. 1999. Dynamic interpretation of verb phrase ellipsis. *Linguistics and Philosophy* 22(2). 187–221. https://doi.org/10.1023/A:1005427813846.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar*. Oxford: Blackwell.

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

Jacobson, Pauline. 2000. Paycheck pronouns, Bach-Peters sentences, and variable-free semantics. *Natural Language Semantics* 8(2). 77–155. https://doi.org/10.1023/A:1026517717879.

Kennedy, Chris. 2014. Predicates *and* formulas: Evidence from ellipsis. In Luka Crnič & Uli Sauerland (eds.), *The art and craft of semantics: A festschrift for Irene Heim*, vol. 1 (MIT Working Papers in Linguistics), 253–277. http://semanticsarchive.net/Archive/jZiNmM4N/.

Kiselyov, Oleg. 2015. Applicative abstract categorial grammars. In Makoto Kanazawa, Lawrence S. Moss & Valeria de Paiva (eds.), *NLCS'15. Third workshop on natural language and computer science*, vol. 32 (EPiC Series), 29–38.

Kobele, Gregory M. 2010. Inverse linking via function composition. *Natural Language Semantics* 18(2). 183–196. https://doi.org/10.1007/s11050-009-9053-7.

McBride, Conor & Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18(1). 1–13. https://doi.org/10.1017/S0956796807006326.

Moggi, Eugenio. 1989. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, 14–23. Pacific Grove, California, USA: IEEE Press. https://doi.org/10.1109/lics.1989.39155.

Muskens, Reinhard. 1995. Tense and the logic of change. In Urs Egli, Peter E. Pause, Christoph Schwarze, Arnim von Stechow & Götz Wienold (eds.), *Lexical Knowledge in the Organization of Language*, 147–183. Amsterdam: John Benjamins. https://doi.org/10.1075/cilt.114.08mus.

Shan, Chung-chieh. 2002. Monads for natural language semantics. In Kristina Striegnitz (ed.), *Proceedings of the ESSLLI 2001 Student Session*, 285–298. http://arxiv.org/abs/cs/0205026.

Shan, Chung-chieh & Chris Barker. 2006. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy* 29(1). 91–134. https://doi.org/10.1007/s10988-005-6580-7.

Sternefeld, Wolfgang. 1998. *The semantics of reconstruction and connectivity*. Arbeitspapier 97, SFB 340. Universität Tübingen & Universität Stuttgart, Germany.

Sternefeld, Wolfgang. 2001. Semantic vs. syntactic reconstruction. In Christian Rohrer, Antje Roßdeutscher & Hans Kamp (eds.), *Linguistic Form and its Computation*, 145–182. Stanford: CSLI Publications.

Wadler, Philip. 1992. Comprehending monads. In *Mathematical Structures in Computer Science*, vol. 2 (special issue of selected papers from 6th Conference on Lisp and Functional Programming), 461–493. https://doi.org/10.1145/91556.91592.

Wadler, Philip. 1995. Monads for functional programming. In Johan Jeuring & Erik Meijer (eds.), *Advanced Functional Programming*, vol. 925 (Lecture Notes in Computer Science), 24–52. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-59451-5_2.