

Static and dynamic alternatives

Computational semantics seminar

February 15, 2017

Reminder about course website

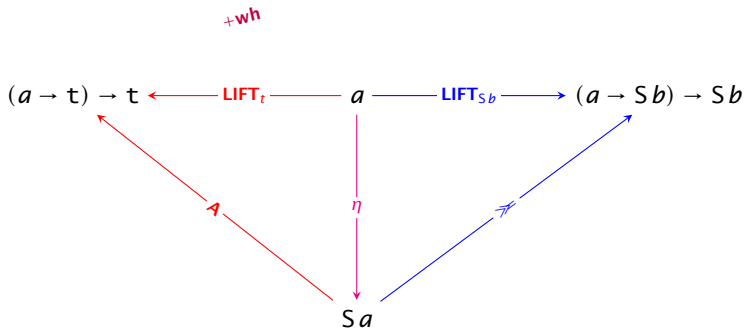
Your one-stop shop for readings, slides, and exercises:

<https://github.com/schar/comp-sem>

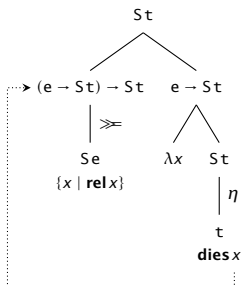
Some of the readings require a password. That password is rutgers.

Basics

The triangle

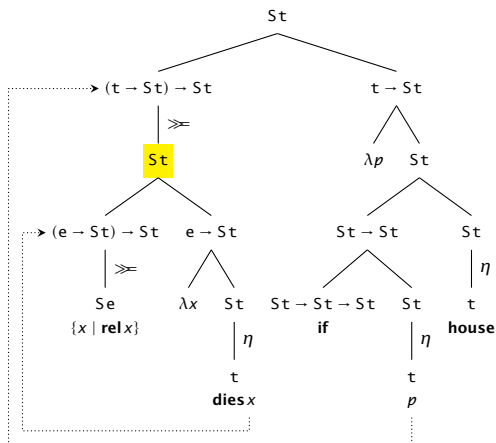


Basic, island derivations



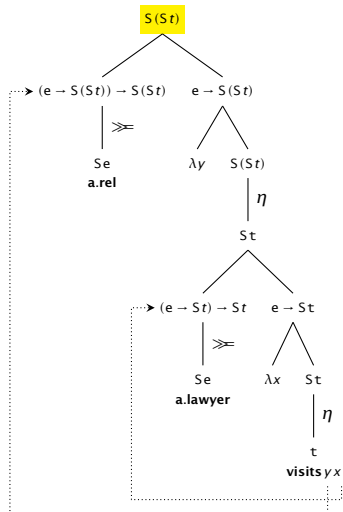
$$= \{\mathbf{dies}\,x \mid \mathbf{rel}\,x\}$$

Basic, island derivations

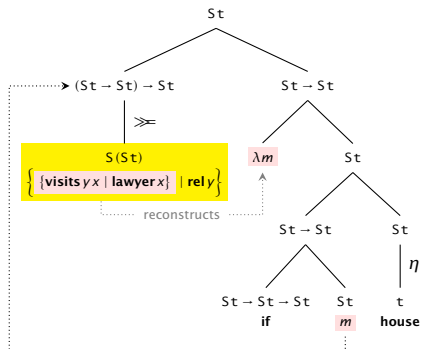


$$= \{dies x \Rightarrow house \mid rel x\}$$

Selectivity via higher-order island meanings



$$= \{ \{ \mathbf{visits} \ yx \mid \mathbf{law} \ x \} \mid \mathbf{rel} \ y \}$$



$$= \{ \mathbf{if} (\exists x \in \mathbf{law} : \mathbf{vis} \ yx) \mathbf{house} \mid \mathbf{rel} \ y \}$$

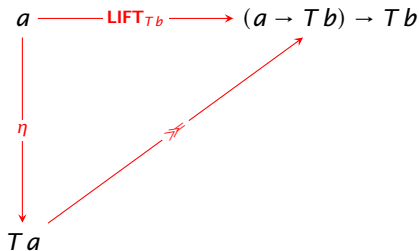
A monad

A **monad** is a type constructor T associated with two functions,
 $\eta :: a \rightarrow T a$ and $\gg= :: T a \rightarrow (a \rightarrow T b) \rightarrow T b$, with three properties:

$$(\eta x) \gg= f = f x \quad (\text{Left Id})$$

$$m \gg= \eta = m \quad (\text{Right Id})$$

$$(m \gg= f) \gg= g = m \gg= \lambda x. f x \gg= g \quad (\text{Assoc})$$



Gives rise to other (weaker notions)

Monadic T 's are **applicative** (i.e., give rise to enriched application):

$$m \gg= \lambda f. n \gg= \lambda x. \eta (f x) = \{f x \mid f \in m, x \in n\}$$

Monadic T 's are **functors** (i.e., give rise to a mapping operation):

$$m \gg= \lambda x. \eta (f x) = \{f x \mid x \in m\}$$

Exercise: (re-)work out what these operations are for S .

- and μ

Every \gg determines (and is determined by) two smaller functions:

$$\begin{aligned}\bullet &:: T a \rightarrow (a \rightarrow b) \rightarrow T b \\ \bullet &:= \lambda m. \lambda f. m \gg \lambda x. \eta (f x)\end{aligned}$$

$$\begin{aligned}\mu &:: T (T a) \rightarrow T a \\ \mu &:= \lambda M. M \gg \lambda m. m\end{aligned}$$

The \bullet is just the mapping operation determined by \gg !

Conversely, $(m \bullet f)^\mu = m \gg f$.

So the two formulations, \gg vs. \bullet and μ , are equivalent. Which one you use is really a matter of preference/convenience.

do notation

Haskell programmers *love* monads, so much so that they invented some special syntax for concisely expressing monadic computations:

► $m \gg= \lambda f. n \gg= \lambda x. \eta (f x)$

```
do f <- m  
    x <- n  
    return (f x)
```

do notation

Haskell programmers *love* monads, so much so that they invented some special syntax for concisely expressing monadic computations:

► $m \gg= \lambda f. n \gg= \lambda x. \eta (f x)$

```
do f <- m
   x <- n
   return (f x)
```

► $m \gg= \lambda x. \eta (f x)$

```
do x <- m
   return (f x)
```

In other words, Haskell programmers are doing very linguist-like things. They're 'QRing' things all over the place!

Binding

Basic data

Indefinites interact with pronouns (obvi!):

1. Indefinites bind pronouns:

A candidate^x submitted her_x paper.

2. Indefinites can be bound into:

A candidate^x submitted a paper she_x had written.

3. ...Including by quantifiers:

No candidate^x submitted a paper she_x had written.

Binding into an island

A key piece of data:

1. Everybody^x loves when [a famous expert on indefinites cites him_x].

$\forall E \gg \forall$

The indefinite can take exceptional scope over the matrix subject. On our account, this should require the [island] to scope over the subject.

Is that scoping compatible with the subject binding into the island?

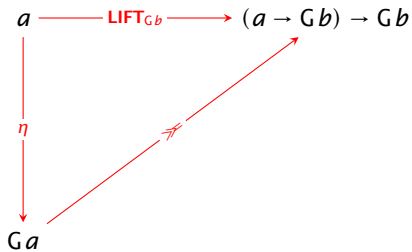
Composition with pronouns

If we work with a standard pronoun semantics, we'll need to extend composition to handle assignment-dependence:

$$\mathbf{she}_0 :: g \rightarrow e$$

$$\mathbf{she}_0 := \lambda g. g_0$$

A binding monad?



A commutative triangle diagram illustrating the relationship between the lifting operation and the monad multiplication. The vertices are a (top-left), Ga (bottom-left), and $(a \rightarrow Gb) \rightarrow Gb$ (top-right). The edges are:

- A horizontal red arrow from a to $(a \rightarrow Gb) \rightarrow Gb$ labeled LIFT_{Gb} in red.
- A vertical red arrow from a down to Ga labeled η in red.
- A diagonal red arrow from Ga up to $(a \rightarrow Gb) \rightarrow Gb$ labeled with a red \rightarrow symbol.

The triangle commutes, indicating that $\text{LIFT}_{Gb} = \eta \circ \rightarrow$.

A binding monad?

Suppose we were going to define a ‘monad instance’ for binding. How would that look?

The type constructor is easy to, ahem, construct:

A binding monad?

Suppose we were going to define a ‘monad instance’ for binding. How would that look?

The type constructor is easy to, ahem, construct:

$$G a := g \rightarrow a$$

And we already know what η would have to be:

A binding monad?

Suppose we were going to define a ‘monad instance’ for binding. How would that look?

The type constructor is easy to, ahem, construct:

$$G a := g \rightarrow a$$

And we already know what η would have to be:

$$\eta x := \lambda g. x$$

What about \gg ?

We know it has to have the type $G a \rightarrow (a \rightarrow G b) \rightarrow G b \dots$

Ditching G for the full blinding glory of the unabbreviated type:

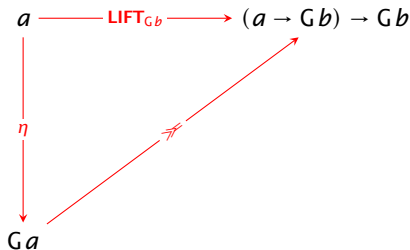
$$(g \rightarrow a) \rightarrow (a \rightarrow g \rightarrow b) \rightarrow g \rightarrow b$$

Can you work out what \gg would have to be?

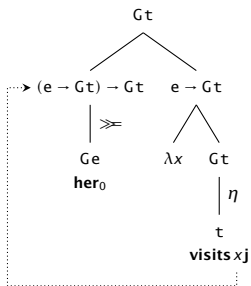
Das it

So that is our monad for binding!

$$\begin{array}{ll} \eta :: a \rightarrow G a & \gg :: G a \rightarrow (a \rightarrow G b) \rightarrow G b \\ \eta := \lambda x. \lambda g. x & \gg := \lambda m. \lambda f. \lambda g. f (m g) g \end{array}$$

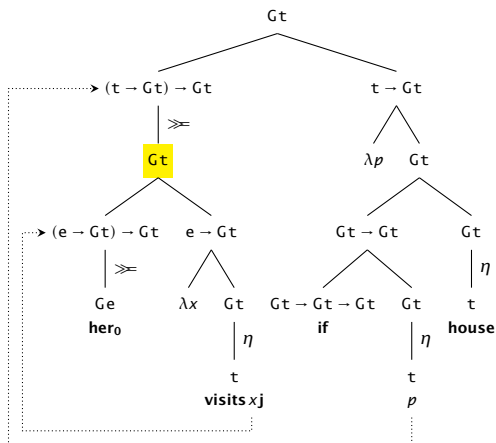


Basic, island derivations



$$= \lambda g. \mathbf{visits}\ g_0\mathbf{j}$$

Basic, island derivations



$$= \lambda g.\text{visits } g_0 j \Rightarrow \text{house}$$

Binding

An operator for effecting binding (cf. Buring 2005):

$$\beta^n :: (a \rightarrow g \rightarrow b) \rightarrow a \rightarrow g \rightarrow b$$

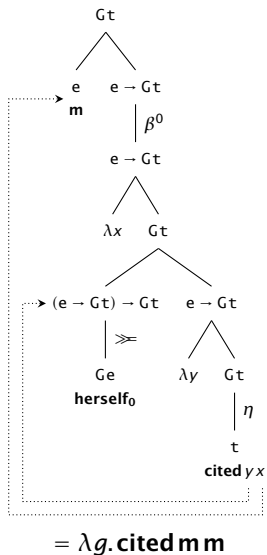
Binding

An operator for effecting binding (cf. Buring 2005):

$$\begin{aligned}\beta^n &:: (a \rightarrow g \rightarrow b) \rightarrow a \rightarrow g \rightarrow b \\ \beta^n &:= \lambda f. \lambda x. \lambda g. f \ x \ g^{n \rightarrow x}\end{aligned}$$

Almost an identity function, but for the fact that it changes the assignment that's fed to f .

Simple binding derivation

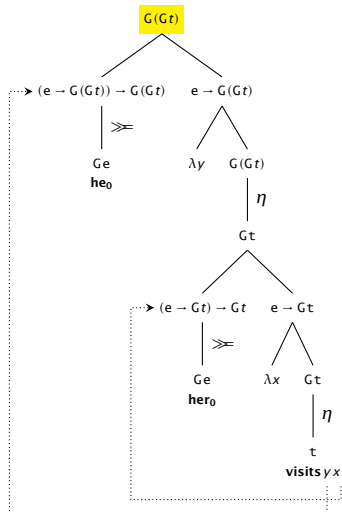


Another parallel

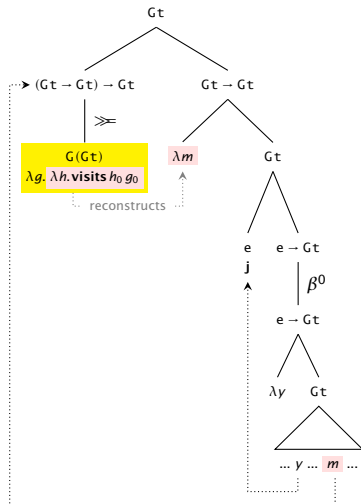
β plays a role rather analogous to closure operations! A pronoun “associated” with a β -operator in roughly the same way an indefinite “associates” with a closure operator.

So what would ‘selective’ association amount to for G ?

Selectivity for G



$$= \lambda g. \lambda h. \mathbf{visits}\ h_0\ g_0$$



$$= \lambda g. \lambda h. \dots \mathbf{visits}\ j\ g_0 \dots$$

Selectivity for G amounts to...

Pronouns with the same index on an island can be fixed to different things outside the island!

Again, the situation is precisely the G analog of selectivity for S!

More generally, any monad will give rise to some notion of selectivity outside islands (though such selectivity will not always be possible to observe in principle; think, e.g., appositive relative clauses).

Is it a monad?

Well, the η and $\gg=$ operations have the right types. So all we need to do is check that they obey **Left** and **Right identity**, and **Associativity**.

Let's just check that **Left identity** is satisfied:

$$(\eta x) \gg= f =$$

Is it a monad?

Well, the η and $\gg=$ operations have the right types. So all we need to do is check that they obey **Left** and **Right identity**, and **Associativity**.

Let's just check that **Left identity** is satisfied:

$$(\eta x) \gg= f = (\lambda g. x) \gg= f$$

Is it a monad?

Well, the η and $\gg=$ operations have the right types. So all we need to do is check that they obey **Left** and **Right identity**, and **Associativity**.

Let's just check that **Left identity** is satisfied:

$$\begin{aligned}(\eta x) \gg= f &= (\lambda g. x) \gg= f \\ &= \lambda h. f ((\lambda g. x) h) h\end{aligned}$$

Is it a monad?

Well, the η and $\gg=$ operations have the right types. So all we need to do is check that they obey **Left** and **Right identity**, and **Associativity**.

Let's just check that **Left identity** is satisfied:

$$\begin{aligned}(\eta x) \gg= f &= (\lambda g. x) \gg= f \\ &= \lambda h. f ((\lambda g. x) h) h \\ &= \lambda h. f x h\end{aligned}$$

Is it a monad?

Well, the η and $\gg=$ operations have the right types. So all we need to do is check that they obey **Left identity**, and **Associativity**.

Let's just check that **Left identity** is satisfied:

$$\begin{aligned}(\eta x) \gg= f &= (\lambda g. x) \gg= f \\&= \lambda h. f ((\lambda g. x) h) h \\&= \lambda h. f x h \\&= f x\end{aligned}$$

Is it a monad?

Well, the η and $\gg=$ operations have the right types. So all we need to do is check that they obey **Left identity**, and **Associativity**.

Let's just check that **Left identity** is satisfied:

$$\begin{aligned}(\eta x) \gg= f &= (\lambda g. x) \gg= f \\&= \lambda h. f ((\lambda g. x) h) h \\&= \lambda h. f x h \\&= f x\end{aligned}\quad \square$$

I won't subject you to the rest of them here, but it's actually not too difficult to prove them on your own. Exercise: try it!

G is applicative

$$m \gg= \lambda f. n \gg= \lambda x. \eta (f x) =$$

G is applicative

$$m \gg= \lambda f. n \gg= \lambda x. \eta (f x) = \lambda g. m g (n g)$$

Does this remind you of anything?

G is applicative

$$m \gg= \lambda f. n \gg= \lambda x. \eta (f x) = \lambda g. m g (n g)$$

Does this remind you of anything? This is simply functional application, enriched with assignments!

Like S (and, indeed, any monad), G is applicative: it supports an enriched form of functional application.

G is a functor

$$m \gg= \lambda x. \eta (f x) =$$

G is a functor

$$m \gg= \lambda x. \eta (f x) = \lambda g. f (m g)$$

Does this remind you of anything?

G is a functor

$$m \gg= \lambda x. \eta (f x) = \lambda g. f (m g)$$

Does this remind you of anything? This is the **mapping** operation for G.
Do you notice anything else?

G is a functor

$$m \gg= \lambda x. \eta (f x) = \lambda g. f (m g)$$

Does this remind you of anything? This is the **mapping** operation for G. Do you notice anything else? It's **function composition**!

Like S (and, indeed, any monad), G is a functor: it supports a mapping operation (if you're familiar with Jacobson's (1999) treatment of pronouns, this might strike you as familiar; hold that thought!).

G 's μ

Can you construct the $\mu :: G(Ga) \rightarrow Ga$ operation for G ?

G's μ

Can you construct the $\mu :: G(Ga) \rightarrow Ga$ operation for G?

$$M^\mu := \lambda g. M g g$$

Layering the two

A tale of 2 monads

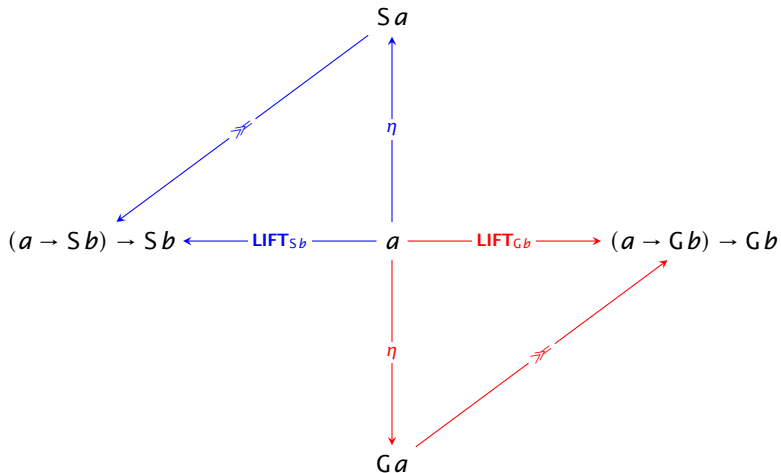
So we have two monads, one for alternatives. . . .

$$\begin{array}{ll} \eta :: a \rightarrow S a & \gg= :: S a \rightarrow (a \rightarrow S b) \rightarrow S b \\ \eta := \lambda p. \{p\} & \gg= := \lambda m. \lambda f. \bigcup_{x \in m} f x \end{array}$$

. . . And one for binding:

$$\begin{array}{ll} \eta :: a \rightarrow G a & \gg= :: G a \rightarrow (a \rightarrow G b) \rightarrow G b \\ \eta := \lambda x. \lambda g. x & \gg= := \lambda m. \lambda f. \lambda g. f (m g) g \end{array}$$

How they [don't] relate...



One way to achieve interaction

Simply layer the monads on top of each other!

she₀ $\gg=$ $\lambda x. \eta$ (**a.ling** $\gg=$ $\lambda y. \eta$ (**visits** $y\ x$))

What does this reduce to?

One way to achieve interaction

Simply layer the monads on top of each other!

$$\mathbf{she}_0 \gg \lambda x. \eta (\mathbf{a.ling} \gg \lambda y. \eta (\mathbf{visits} \ y \ x))$$

What does this reduce to?

$$\lambda g. \{ \mathbf{visits} \ y \ g_0 \mid \mathbf{ling} \ x \} :: G(\mathbf{St})$$

Binding by indefinites

Can be achieved as well by layering:

$$\mathbf{a.ling} \gg \lambda x. \eta (\dots x (\lambda y. \lambda g. \dots)^{\beta_0} \dots)$$

Leads to something with the following shape:

$$\{\lambda g. \dots x \dots x \dots \mid \mathbf{ling} x\} :: S(Gt)$$

Lots of higher-order structure

Possibly problematic:

1. A linguist^x submitted a paper she_x wrote.

Here, we'll inevitably end up with alternatives over assignments over alternatives: $S(G(St))$. That's a lot of layering! And in principle, we'll need arbitrarily deep layerings of types (e.g., if the second indefinite does some binding of its own).

Oy!

A choice

We might wish to derive a *combined* monad so that we might walk and chew gum at the same time.

But as with alternative semantics, this presents us with a choice. Do we layer binding around alternatives, or alternatives around binding?

$$GS\ a ::= g \rightarrow S\ a \qquad SG\ a := S\ (g \rightarrow a)$$

But for bind?

It turns out (I think!) that only GS supports a $\gg=$ operation:

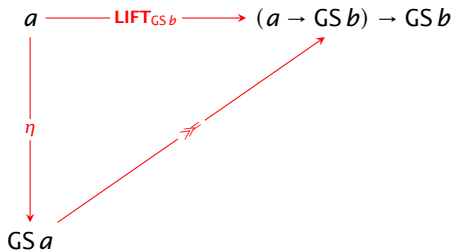
$$m \gg= f := \lambda g. \bigcup_{x \in mg} f x g$$

Compared to our previous operation, this is just adding some g 's:

$$m \gg= f := \bigcup_{x \in m} f x$$

So only (I think!) one layering works.

Yet another triangle



Rounding out the picture

Meanings for indefinites:

Rounding out the picture

Meanings for indefinites:

$$\mathbf{a.ling} := \lambda g. \{x \mid \mathbf{ling} x\}$$

Meanings for pronouns:

Rounding out the picture

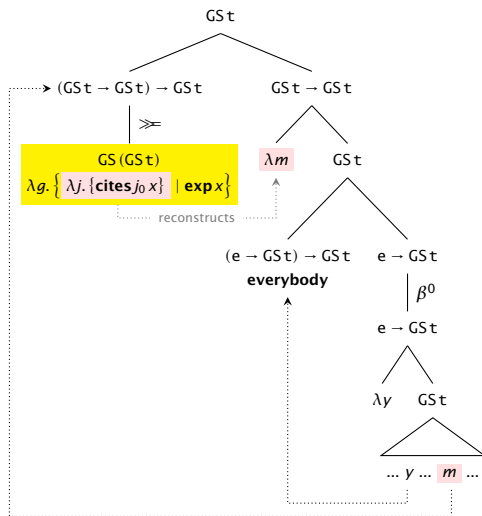
Meanings for indefinites:

$$\mathbf{a.ling} := \lambda g. \{x \mid \mathbf{ling} x\}$$

Meanings for pronouns:

$$\mathbf{she}_0 := \lambda g. \{g_0\}$$

Higher-order derivations



Dynamics

Basic data

Indefinites look more like proper names than quantifiers with respect to discourse anaphora:

1. {Polly, a linguist, *every linguist, *no linguist}ⁿ entered. She_n sat.

Such data is difficult to explain if we treat indefinites as quantifiers!

The standard account

In **dynamic semantics**, sentence meanings store values for variables:

$$\llbracket \text{Polly}^n \text{ entered} \rrbracket = \lambda g. \{g^{n \rightarrow \mathbf{p}} \mid \mathbf{entered} \mathbf{p}\}$$

$$\llbracket \text{a linguist}^n \text{ entered} \rrbracket = \lambda g. \{g^{n \rightarrow x} \mid \mathbf{entered} x\}$$

Type: $g \rightarrow Sg$

Compare the corresponding static meanings:

$$\llbracket \text{Polly}^n \text{ entered} \rrbracket = \lambda g. \mathbf{entered} \mathbf{p}$$

$$\llbracket \text{a linguist}^n \text{ entered} \rrbracket = \lambda g. \mathbf{entered} x$$

Type: $g \rightarrow t$

Pictorially

$g \dashrightarrow \text{Polly}^n \text{ entered} \dashrightarrow g^{n \rightarrow p}$

$g \dashrightarrow \text{a linguist}^n \text{ entered} \dashrightarrow \begin{matrix} g^{n \rightarrow L_1} \\ g^{n \rightarrow L_2} \\ g^{n \rightarrow L_3} \\ g^{n \rightarrow L_4} \\ g^{n \rightarrow L_5} \end{matrix}$

A dynamic monad

Fixing a type constructor:

$$D a ::= g \rightarrow S(a \times g)$$

With a corresponding η operation:

$$\eta x := \lambda g. \{(x, g)\}$$

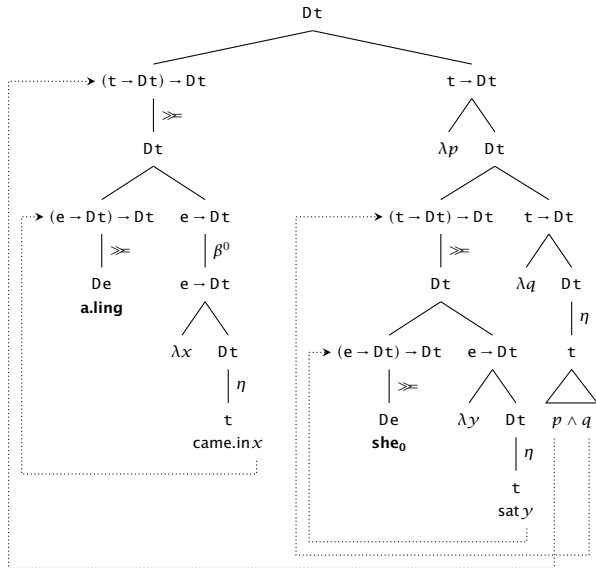
... And a corresponding $\gg=$ operation:

$$m \gg= f := \lambda g. \bigcup_{(x, h) \in m} f x h$$

Summing up

$T a$	ηx	m^{\gg}	a.ling	she₀
$S a$	$\{x\}$	$\lambda f. \bigcup_{x \in m} f x$	$\{x \mid \mathbf{ling} x\}$	—
$g \rightarrow S a$	$\lambda g. \{x\}$	$\lambda f. \lambda g. \bigcup_{x \in mg} f x g$	$\lambda g. \{x \mid \mathbf{ling} x\}$	$\lambda g. \{g_0\}$
$g \rightarrow S(a \times g)$	$\lambda g. \{(x, g)\}$	$\lambda f. \lambda g. \bigcup_{(x, h) \in mg} f x h$	$\lambda g. \{(x, g) \mid \mathbf{ling} x\}$	$\lambda g. \{(g_0, g)\}$

A derivation



Applicatives vs. functors vs. monads

Composition for η

For either layering, it's straightforward to derive the corresponding η :

$$\eta_{GS} x :=$$

Composition for η

For either layering, it's straightforward to derive the corresponding η :

$$\eta_{\text{GS}} x := \lambda g. \{x\}$$

Composition for η

For either layering, it's straightforward to derive the corresponding η :

$$\eta_{GS} x := \lambda g. \{x\} \qquad \eta_{SG} x :=$$

Composition for η

For either layering, it's straightforward to derive the corresponding η :

$$\eta_{GS} x := \lambda g. \{x\}$$

$$\eta_{SG} x := \{\lambda g. x\}$$

You'll notice that each of these operations is a *composition* of η_S and η_G (in different orders):

$$\eta_{GS} \equiv \eta_G \circ \eta_S$$

$$\eta_{SG} \equiv \eta_S \circ \eta_G$$

So *this* much, at least, is automatic. Whichever option we choose, we'll have a ready and waiting η operation.

Applicatives and functors

GS and SG are both applicative:

$$\lambda g. \{f\,x \mid f \in mg, x \in ng\} \quad \{\lambda g. f\,g(x\,g) \mid f \in m, x \in n\}$$

GS and SG are both functors:

$$\lambda g. \{f\,x \mid x \in mg\} \quad \{\lambda g. f\,(x\,g) \mid x \in m\}$$

Generating a functor from two monads

```
fmap_ :: (Monad m, Monad n) =>  
      (a -> b) -> m (n a) -> m (n b)
```

```
fmap_ f mx = do nx <- mx  
              return (do x <- nx  
                        return (f x))
```

Generating an applicative from two monads

```
app_ :: (Monad m, Monad n) =>  
      m (n (a -> b)) -> m (n a) -> m (n b)
```

```
app_ mnf mnx = do nf <- mnf  
                  nx <- mnx  
                  return (do f <- nf  
                              x <- nx  
                              return (f x))
```

Generating a monad from two monads?

```
hmm :: (Monad m, Monad n) =>  
      m (n a) -> (a -> m (n b)) -> m (n (m (n b)))
```

```
mnx `hmm` f = do nx <- mnx  
                return (do x <- nx  
                           return (f x))
```

Our specific case

```
swap :: [i -> a] -> i -> [a]
swap m = \g -> [f g | f <- m]
```

```
bind :: (i -> [a]) -> (a -> i -> [b]) -> i -> [b]
mnmnx `bind` f = do nmnx <- mnmnx `hmm` f
                   nnx <- swap nmnx
                   return (concat nnx)
```

Büring, Daniel. 2005. *Binding theory*. New York: Cambridge University Press.
<http://dx.doi.org/10.1017/cbo9780511802669>.

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