Effectful composition in natural language semantics

Ups and downs: adjunctions, (co)monads, and association with effects

Dylan Bumford (UCLA) Simon Charlow (Rutgers)

ESSLLI 2022, NUI Galway

Recap

Denotations via functors

Expression	Type	Denotation
no cat	$\mathtt{C\,e} ::= (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$	$\lambda c.\, \neg\exists x.\, \mathbf{cat}\, x \wedge c\, x$
the cat	$\mathtt{M\,e} ::= \mathtt{e} \mid \#$	x if $\mathbf{cat} = \{x\}$ else $\#$
Sassy, a cat	$\mathtt{W\,e} ::= \mathtt{e} \times \mathtt{t}$	$\langle \mathbf{s}, \mathbf{cat}\, \mathbf{s} \rangle$
she	$\mathtt{R\,e} ::= \mathtt{r} \to \mathtt{e}$	$\lambda g.\, g_0$
which cat	$\mathtt{S\,e} ::= \{\mathtt{e}\}$	$\{x \mid \mathbf{cat}\, x\}$
SASSY	$\mathtt{F\,e} ::= \mathtt{e} \times \{\mathtt{e}\}$	$\langle \mathbf{s}, \{x \mid x \in D_e\} \rangle$
a cat	$\mathtt{D\,e} ::= \mathtt{s} \to \{\mathtt{e} \times \mathtt{s}\}$	$\lambda s.\, \{\langle x, s \mathbin{+\!\!+} x \rangle \mid \mathbf{cat}\, x\}$
...	...	...

Meditate on the hoops you'd need to jump through to develop a theory of grammar in the standard mold that could handle all these effects (and more).

Ascending typeclasses

We have explored a hierarchy of abstractions for modeling linguistic side effects:

```haskell
-- Functors: as many layers as effectful things
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- Applicatives: contexts can be merged
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

-- Monads: higher-order contexts can be flattened
class Applicative f => Monad f where
  join :: f (f a) -> f a -- or
  (>>=) :: f a -> (a -> f b) -> f b
```

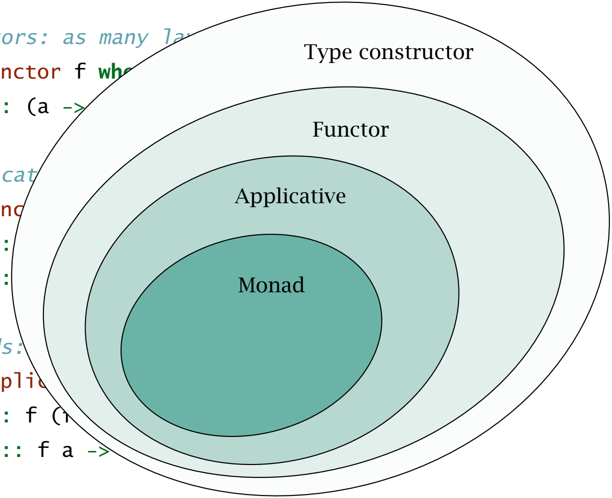
Ascending typeclasses

We have explored a hierarchy of abstractions for modeling linguistic side effects:

```
-- Functors: as many la
class Functor f whe
  fmap :: (a ->

-- Applicat
class Func
  pure ::
  (<*>) :

-- Monads:
class Applic
  join :: f (f
  (>>=) :: f a ->
```



Some examples

Here's a type constructor that's not a functor: (where's $\bullet$?)

- $\mathsf{X}\,a ::= a \rightarrow \mathsf{r}$

Here's a functor that's not an applicative: (where's η? where's $\circledast$?)

- $\mathsf{Y}\,a ::= a \times \mathsf{e}$

Here's a couple applicatives that (probably) aren't monads: (where's μ?)

- $\mathsf{SR}\,a ::= \{\mathsf{r} \rightarrow a\}$
- $\mathsf{WR}\,a ::= (\mathsf{r} \rightarrow a) \times m$ (m a monoid)

Implemented by extending type-driven semantic parsing

$$\text{if } a \cdot b \Rightarrow (f, c)\text{, then} \begin{cases} F\,a \cdot \,b \Rightarrow (\uparrow\!\mathbf{R}\,f\,l\,r \coloneqq (\lambda l'.\,f\,l'\,r) \bullet l,\ F\,c) \\ \,a \cdot F\,b \Rightarrow (\uparrow\!\mathbf{L}\,f\,l\,r \coloneqq (\lambda r'.\,f\,l\,r') \bullet r,\ F\,c) \\ F\,a \cdot F\,b \Rightarrow (\ \mathbf{A}\ f\,l\,r \coloneqq f \bullet l \circledast r,\qquad F\,c) \end{cases}$$

To these binary rules, we can add monadic `join`-ing:

$$\text{if } a \cdot b \Rightarrow (f, MM\,c)\text{, then } a \cdot b \Rightarrow (\mathbf{J}\,f\,l\,r \coloneqq \mu\,(f\,l\,r), M\,c)$$

Functors compose

$$(\bullet)\,((\bullet)\,f) :: F\,(G\,a) \to F\,(G\,b)$$

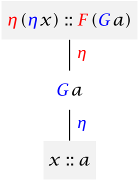
$$\Big|\;\bullet$$

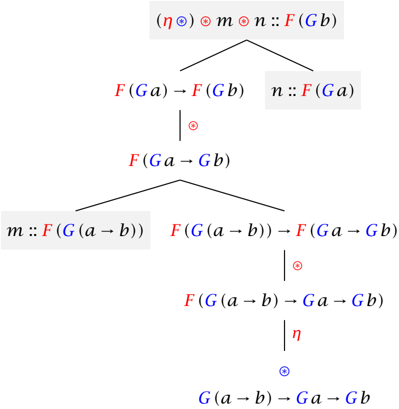
$$(\bullet)\,f :: G\,a \to G\,b$$

$$\Big|\;\bullet$$

$$f :: a \to b$$

Applicative functors compose





Monads don't always compose

$$\eta\,(\eta\,x) :: F\,G\,a$$

$$\Big|\ \eta$$

$$G\,a$$

$$\Big|\ \eta$$

$$x :: a$$

$$F\,G\,a$$

$$F\,G\,G\,a \qquad F\,G\,G\,a \to F\,G\,a$$

$$\Big|\ \mu \qquad\qquad \Big|\ \bullet$$

$$F\,F\,G\,G\,a \qquad \mu :: G\,G\,a \to G\,a$$

$$\vdots\ F(\Upsilon)?$$

$$F\,G\,F\,G\,b$$

$$m :: F\,G\,a \qquad F\,G\,a \to F\,G\,F\,G\,b$$

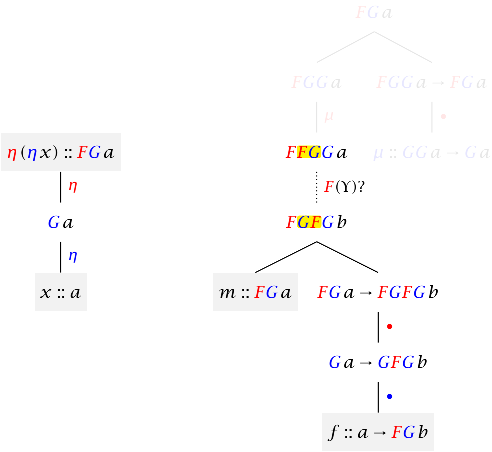
$$\Big|\ \bullet$$

$$G\,a \to G\,F\,G\,b$$

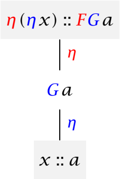
$$\Big|\ \bullet$$

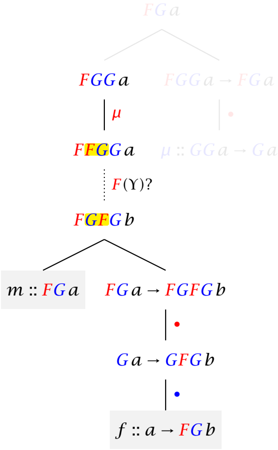
$$f :: a \to F\,G\,b$$

Monads don’t always compose



Monads don't always compose

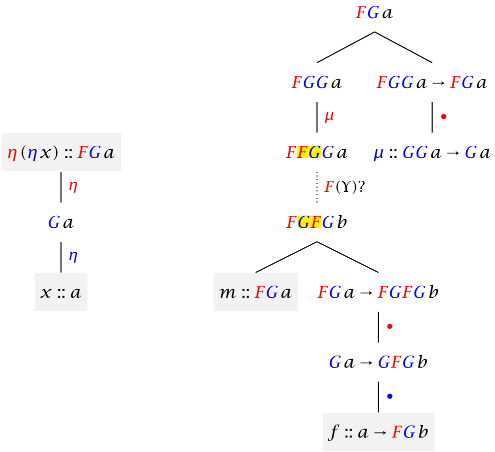




Monads don’t always compose

$$\begin{array}{c} \eta\,(\eta\,x) :: F\,G\,a \\ \Big|\;\eta \\ G\,a \\ \Big|\;\eta \\ x :: a \end{array}
\qquad\qquad
\begin{array}{c}
F\,G\,a \\
\diagup \qquad\qquad \diagdown \\
\begin{array}{c} F\,G\,G\,a \\ \Big|\;\mu \\ F\,F\,G\,G\,a \\ \vdots\;F\,(\Upsilon)? \\ F\,G\,F\,G\,b \\ \diagup \qquad\qquad \diagdown \\ m :: F\,G\,a \qquad \begin{array}{c} F\,G\,a \to F\,G\,F\,G\,b \\ \Big|\;\bullet \\ G\,a \to G\,F\,G\,b \\ \Big|\;\bullet \\ f :: a \to F\,G\,b \end{array} \end{array}
\qquad
\begin{array}{c} F\,G\,G\,a \to F\,G\,a \\ \Big|\;\bullet \\ \mu :: G\,G\,a \to G\,a \\ \\ \\ \\ \\ \\ \end{array}
\end{array}$$

Monads don’t always compose



…BUT!

Transformers (Liang, Hudak & Jones 1995). For any monadic $(M, \eta, \star)$:

ReaderT: adding env-sensitivity

- $\mathsf{RT}\, M\, a ::= \mathsf{r} \to M\, a$
- $\eta\, x := \lambda r.\, \eta\, x$
- $m \star f = \lambda r.\, m\, r \star \lambda x.\, f\, x\, r$

...BUT!

Transformers (Liang, Hudak & Jones 1995). For any monadic $(M, \eta, \star)$:

ReaderT: adding env-sensitivity

- $\mathsf{RT}\, M\, a ::= \mathsf{r} \rightarrow M\, a$
- $\eta\, x := \lambda r.\, \eta\, x$
- $m \star f = \lambda r.\, m\, r \star \lambda x.\, f\, x\, r$

StateT: adding state

- $\mathsf{ST}\, M\, a ::= \mathsf{s} \rightarrow M\, (a \times \mathsf{s})$
- $\eta\, x = \lambda s.\, \eta\, (x, s)$
- $m \star f = \lambda s.\, m\, s \star \lambda (x, s').\, f\, x\, s'$

...BUT!

Transformers (Liang, Hudak & Jones 1995). For any monadic $(M, \eta, \star)$:

ReaderT: adding env-sensitivity

- $\mathsf{RT}\,M\,a ::= \mathsf{r} \to M\,a$
- $\eta\,x := \lambda r.\,\eta\,x$
- $m \star f = \lambda r.\,m\,r \star \lambda x.\,f\,x\,r$

ContT: adding scope

- $\mathsf{CT}\,M\,a ::= (a \to M\,\mathsf{t}) \to M\,\mathsf{t}$
- $\eta\,x = \lambda k.\,k\,x$
- $m \star f = \lambda k.\,m\,(\lambda x.\,f\,x\,k)$

StateT: adding state

- $\mathsf{ST}\,M\,a ::= \mathsf{s} \to M\,(a \times \mathsf{s})$
- $\eta\,x = \lambda s.\,\eta\,(x, s)$
- $m \star f = \lambda s.\,m\,s \star \lambda(x, s').\,f\,x\,s'$

. . . BUT!

Transformers (Liang, Hudak & Jones 1995). For any monadic $(M, \eta, \star)$:

ReaderT: adding env-sensitivity

- $\mathtt{RT}\,M\,a ::= \mathtt{r} \to M\,a$
- $\eta\,x := \lambda r.\,\eta\,x$
- $m \star f = \lambda r.\,m\,r \star \lambda x.\,f\,x\,r$

ContT: adding scope

- $\mathtt{CT}\,M\,a ::= (a \to M\,\mathtt{t}) \to M\,\mathtt{t}$
- $\eta\,x = \lambda k.\,k\,x$
- $m \star f = \lambda k.\,m\,(\lambda x.\,f\,x\,k)$

StateT: adding state

- $\mathtt{ST}\,M\,a ::= \mathtt{s} \to M\,(a \times \mathtt{s})$
- $\eta\,x = \lambda s.\,\eta\,(x, s)$
- $m \star f = \lambda s.\,m\,s \star \lambda(x, s').\,f\,x\,s'$

The **Identity** monad

- $\mathtt{I}\,a ::= a$
- $\eta\,a = a$
- $m \star f = f\,m$

It's spectacular, and a bit eerie, to notice that CT differs from C only in its monadic return type (Wadler 1994, Charlow 2014).

The higher-order

Composition with applicatives and monads

Days 3 and 4: applicatives/monads for avoiding/escaping the higher-order…

```haskell
-- Applicatives: contexts can be merged
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

-- Monads: higher-order contexts can be flattened
class Applicative f => Monad f where
  join :: f (f a) -> f a -- or
  (>>=) :: f a -> (a -> f b) -> f b
```

Higher-order effects: Writing

```
saw   = [("saw"          , TV, E :-> E :-> T)]
maling = [("Mary--a ling", DP, effW T E) ]
sacat  = [("Sassy--a cat", DP, effW T E) ]
```

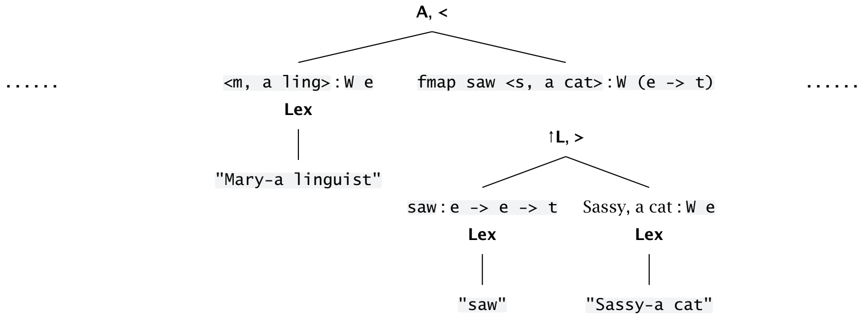
saw :: e → e → t

Mary, a linguist :: W e

Sassy, a cat :: W e

```
GHCi> parse $ [maling, saw, sacat]
```

```
(<*>) (fmap (\l -> (\r -> r l)) <m, a ling>) (fmap saw <s, a cat>) : W t
```



Higher-order effects: Writing

```
saw    = [("saw"           , TV, E :-> E :-> T)]
maling = [("Mary--a ling", DP, effW T E) ]
sacat  = [("Sassy--a cat", DP, effW T E) ]
```

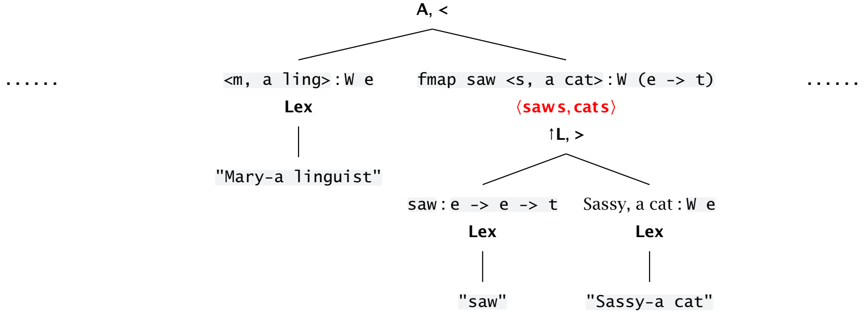
saw :: e → e → t

Mary, a linguist :: W e

Sassy, a cat :: W e

```
GHCi> parse $ [maling, saw, sacat]
```

```
(<*>) (fmap (\l -> (\r -> r l)) <m, a ling>) (fmap saw <s, a cat>) : W t
```



Higher-order effects: Writing

```
saw    = [("saw"          , TV, E :-> E :-> T)]
maling = [("Mary--a ling", DP, effW T E) ]
sacat  = [("Sassy--a cat", DP, effW T E) ]
```

saw :: e → e → t

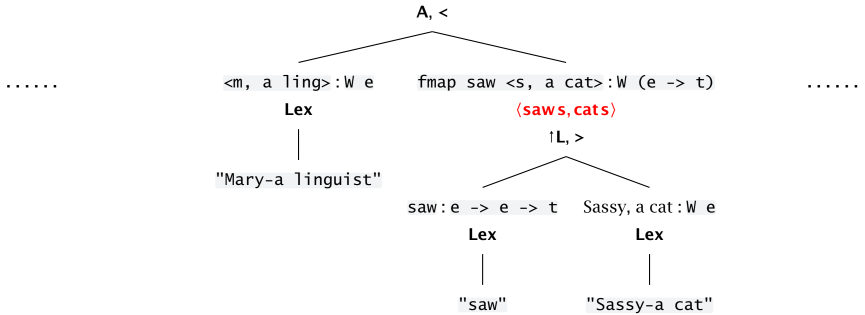
Mary, a linguist :: W e

Sassy, a cat :: W e

```
GHCi> parse $ [maling, saw, sacat]
```

```
(<*>) (fmap (\l -> (\r -> r l)) <m, a ling>) (fmap saw <s, a cat>) : W t
```

$$= \langle \text{LIFT}\, \mathbf{m}, \mathbf{ling\, m}\rangle \circledast \langle \mathbf{saw\, s}, \mathbf{cat\, s}\rangle$$



Higher-order effects: Writing

```
saw    = [("saw"           , TV, E :-> E :-> T)]
maling = [("Mary--a ling", DP, effW T E) ]
sacat  = [("Sassy--a cat", DP, effW T E) ]
```

saw :: e → e → t

Mary, a linguist :: W e

Sassy, a cat :: W e

```
GHCi> parse $ [maling, saw, sacat]
```

`(<*>) (fmap (\l -> (\r -> r l)) <m, a ling>) (fmap saw <s, a cat>)` : `W t`

$= \langle \text{LIFT}\, \mathbf{m}, \mathbf{ling\,m} \rangle \circledast \langle \mathbf{saw\,s}, \mathbf{cat\,s} \rangle$

$= \langle \mathbf{saw\,s\,m}, \mathbf{ling\,m} \wedge \mathbf{cat\,s} \rangle$

A, <

...... `<m, a ling>` : `W e` — `fmap saw <s, a cat>` : `W (e -> t)` ......

Lex — $\langle \mathbf{saw\,s}, \mathbf{cat\,s} \rangle$

`"Mary-a linguist"` — **↑L, >**

`saw` : `e -> e -> t` — Sassy, a cat : `W e`

Lex — **Lex**

`"saw"` — `"Sassy-a cat"`

Higher-order effects: Writing

```
fmap (\x -> fmap (\f -> f x) (fmap saw <s, a cat>)) <m, a ling>:W (W t)
```

↑R, ↑L, <

`<m, a ling>:W e`

Lex

`"Mary-a linguist"`

`fmap saw <s, a cat>:W (e -> t)`

↑L, >

`saw:e -> e -> t`

Lex

`"saw"`

`<s, a cat>:W e`

Lex

`"Sassy-a cat"`

Higher-order effects: Writing

`fmap (\x -> fmap (\f -> f x) (fmap saw <s, a cat>)) <m, a ling>` : `W (W t)`

$(\lambda x.\,(\mathrm{LIFT}\,x) \bullet \langle \mathbf{saw\,s}, \mathbf{cat\,s}\rangle) \bullet \langle \mathbf{m}, \mathbf{ling\,m}\rangle$

↑R, ↑L, <

`<m, a ling>` : `W e`   `fmap saw <s, a cat>` : `W (e -> t)`

Lex   **↑L, >**

`"Mary-a linguist"`   `saw` : `e -> e -> t`   `<s, a cat>` : `W e`

Lex   **Lex**

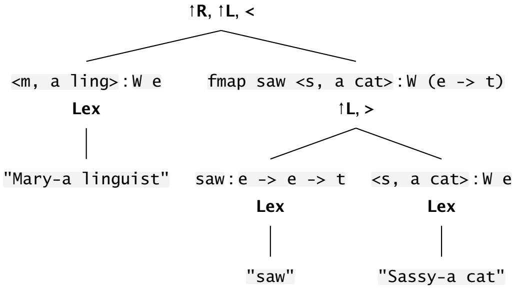
`"saw"`   `"Sassy-a cat"`

Higher-order effects: Writing

```
fmap (\x -> fmap (\f -> f x) (fmap saw <s, a cat>)) <m, a ling> : W (W t)
```

$$(\lambda x.\,(\text{LIFT}\,x) \bullet \langle \mathbf{saw\,s}, \mathbf{cat\,s}\rangle) \bullet \langle \mathbf{m}, \mathbf{ling\,m}\rangle$$

$$(\lambda x.\,\langle \mathbf{saw\,s}\,x, \mathbf{cat\,s}\rangle) \bullet \langle \mathbf{m}, \mathbf{ling\,m}\rangle$$



Higher-order effects: Writing

fmap (\x -> fmap (\f -> f x) (fmap saw <s, a cat>)) <m, a ling> : W (W t)

$(\lambda x.\,(\text{LIFT}\,x) \bullet \langle \mathbf{saw\,s}, \mathbf{cat\,s}\rangle) \bullet \langle \mathbf{m}, \mathbf{ling\,m}\rangle$

$(\lambda x.\,\langle \mathbf{saw\,s}\,x, \mathbf{cat\,s}\rangle) \bullet \langle \mathbf{m}, \mathbf{ling\,m}\rangle$

$\langle\langle \mathbf{saw\,s\,m}, \mathbf{cat\,s}\rangle, \mathbf{ling\,m}\rangle$

↑R, ↑L, <

<m, a ling> : W e

Lex

"Mary-a linguist"

fmap saw <s, a cat> : W (e -> t)

↑L, >

saw : e -> e -> t

Lex

"saw"

<s, a cat> : W e

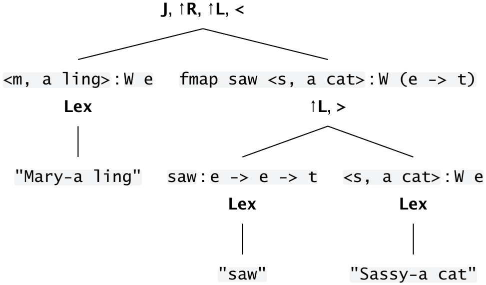
Lex

"Sassy-a cat"

Higher-order effects: Writing

- Recall that $\mu_{\mathtt{W}}\langle\langle a,p\rangle,q\rangle = \langle a, q \wedge p\rangle$

```
join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw <s, a cat>)) <m, a ling>):W t
```

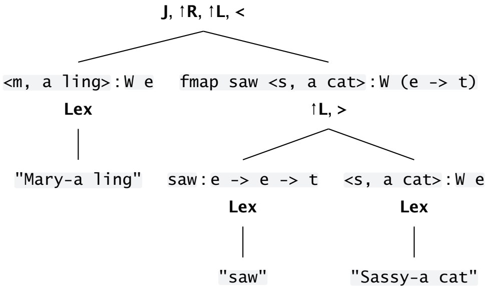


Higher-order effects: Writing

- Recall that $\mu_{\mathsf{W}}\,\langle\langle a, p\rangle, q\rangle = \langle a, q \wedge p\rangle$

```
join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw <s, a cat>)) <m, a ling>) : W t
```

$= \mu\,\langle\langle \mathbf{saw\,s\,m}, \mathbf{cat\,s}\rangle, \mathbf{ling\,m}\rangle$



Higher-order effects: Writing

- Recall that $\mu_{\mathtt{W}}\langle\langle a, p\rangle, q\rangle = \langle a, q \wedge p\rangle$

`join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw <s, a cat>)) <m, a ling>)` : `W t`

$= \mu\langle\langle \mathbf{saw\,s\,m}, \mathbf{cat\,s}\rangle, \mathbf{ling\,m}\rangle$

$= \langle \mathbf{saw\,s\,m}, \mathbf{ling\,m} \wedge \mathbf{cat\,s}\rangle$

J, ↑**R**, ↑**L**, **<**

`<m, a ling>` : `W e` `fmap saw <s, a cat>` : `W (e -> t)`

Lex ↑**L**, **>**

`"Mary-a ling"` `saw` : `e -> e -> t` `<s, a cat>` : `W e`

Lex **Lex**

`"saw"` `"Sassy-a cat"`

Higher-order effects: Just an annoying detour?

```
join (fmap (\a -> fmap (\f -> f a) (fmap saw <s, a cat>)) <m, a ling>) : W t
```

$= \langle \mathbf{saw\,s\,m}, \mathbf{ling\,m} \wedge \mathbf{cat\,s} \rangle$

$\mathbf{J}, \uparrow\mathbf{R}, \uparrow\mathbf{L}, \mathbf{<}$

```
<m, a ling> : W e        fmap saw <s, a cat> : W (e -> t)
```

```
(<*>) (fmap (\l -> (\r -> r l)) <m, a ling>) (fmap saw <s, a cat>) : W t
```

$= \langle \mathbf{saw\,s\,m}, \mathbf{ling\,m} \wedge \mathbf{cat\,s} \rangle$

$\mathbf{A}, \mathbf{<}$

```
<m, a ling> : W e        fmap saw <s, a cat> : W (e -> t)
```

Higher-order effects: Continuations

Let's try another one.

```
saw      = [("saw"     , TV, E :-> E :-> T)]
someone  = [("someone" , DP, effC T T E)   ]
everyone = [("everyone", DP, effC T T E)   ]
```

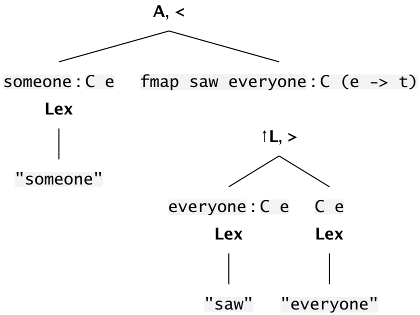
saw :: e → e → t

someone :: C e

everyone :: C e

```
GHCi> parse $ [someone, saw, everyone]
```

`(<*>) (fmap (\x -> (\f -> f x)) someone) (fmap saw everyone)` : `C t`



Higher-order effects: Continuations

Let's try another one.

```
saw      = [("saw"     , TV, E :-> E :-> T)]
someone  = [("someone" , DP, effC T T E)   ]
everyone = [("everyone", DP, effC T T E)   ]
```

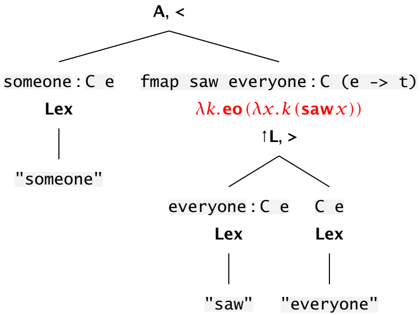
saw :: e → e → t

someone :: C e

everyone :: C e

```
GHCi> parse $ [someone, saw, everyone]
```

`(<*>) (fmap (\x -> (\f -> f x)) someone) (fmap saw everyone)` : `C t`



Higher-order effects: Continuations

Let's try another one.

```
saw      = [("saw"     , TV, E :-> E :-> T)]
someone  = [("someone" , DP, effC T T E)   ]
everyone = [("everyone", DP, effC T T E)   ]
```

saw :: e → e → t

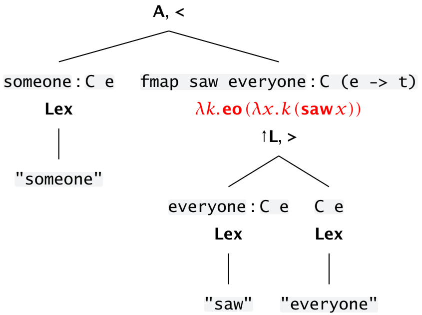
someone :: C e

everyone :: C e

```
GHCi> parse $ [someone, saw, everyone]
```

`(<*>) (fmap (\x -> (\f -> f x)) someone) (fmap saw everyone)` : `C t`

$= (\lambda k.\, \mathbf{so}\, (\lambda y.\, k\, (\textsc{lift}\, y))) \circledast (\lambda k.\, \mathbf{eo}\, (\lambda y.\, k\, (\mathbf{saw}\, x)))$



Higher-order effects: Continuations

Let's try another one.

```
saw       = [("saw"     , TV, E :-> E :-> T)]
someone   = [("someone" , DP, effC T T E)   ]
everyone  = [("everyone", DP, effC T T E)   ]
```

saw :: e → e → t

someone :: C e

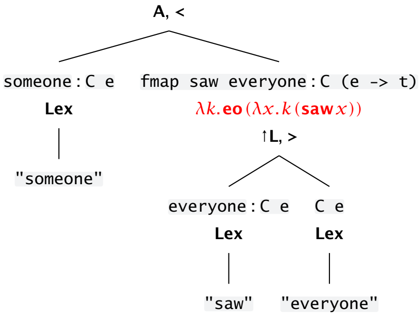
everyone :: C e

```
GHCi> parse $ [someone, saw, everyone]
```

`(<*>) (fmap (\x -> (\f -> f x)) someone) (fmap saw everyone)` : `C t`

$$= (\lambda k.\,\mathbf{so}\,(\lambda y.\,k\,(\textsc{lift}\,y))) \circledast (\lambda k.\,\mathbf{eo}\,(\lambda y.\,k\,(\mathbf{saw}\,x)))$$

$$= \lambda k.\,\mathbf{so}\,(\lambda y.\,\mathbf{eo}\,(\lambda x.\,k\,(\mathbf{saw}\,x\,y)))$$



Higher-order effects: Continuations

`fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone : C (C t)`

↑R, ↑L, <

`someone : C e`

Lex

`"someone"`

`fmap saw everyone : C (e -> t)`

$\lambda k.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x))$

↑L, >

`saw : e -> e -> t`

Lex

`"saw"`

`everyone : C e`

Lex

`"everyone"`

Higher-order effects: Continuations

`fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone` : `C (C t)`

$$= (\lambda k.\mathbf{so}\,(\lambda y.k\,((\textsc{lift}\,y) \bullet (\lambda c.\mathbf{eo}\,(\lambda x.c\,(\mathbf{saw}\,x))))))$$

↑R, ↑L, <

`someone` : `C e`

Lex

`"someone"`

`fmap saw everyone` : `C (e -> t)`

$\lambda k.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x))$

↑L, >

`saw` : `e -> e -> t`

Lex

`"saw"`

`everyone` : `C e`

Lex

`"everyone"`

Higher-order effects: Continuations

`fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone` : `C (C t)`

$= (\lambda k.\,\mathbf{so}\,(\lambda y.\,k\,((\text{LIFT}\,y) \bullet (\lambda c.\,\mathbf{eo}\,(\lambda x.\,c\,(\mathbf{saw}\,x))))))$

$= (\lambda k.\,\mathbf{so}\,(\lambda y.\,k\,(\lambda c.\,\mathbf{eo}\,(\lambda x.\,c\,(\mathbf{saw}\,x\,y)))))$

↑R, ↑L, <

`someone` : `C e`

Lex

`"someone"`

`fmap saw everyone` : `C (e -> t)`

$\lambda k.\,\mathbf{eo}\,(\lambda x.\,k\,(\mathbf{saw}\,x))$

↑L, >

`saw` : `e -> e -> t`

Lex

`"saw"`

`everyone` : `C e`

Lex

`"everyone"`

Higher-order effects: Continuations

- Recall $\mu_{\mathsf{C}}\, M = \lambda k. M\, (\lambda m. m\, k)$

`join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone)` : `C t`

J, ↑R, ↑L, <

`someone` : `C e` — **Lex** — `"someone"`

`fmap saw everyone` : `C (e -> t)`

$\lambda k.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x))$

↑L, >

`saw` : `e -> e -> t` — **Lex** — `"saw"`

`everyone` : `C e` — **Lex** — `"everyone"`

Higher-order effects: Continuations

- Recall $\mu_{\mathsf{C}}\, M = \lambda k. M\,(\lambda m. m\, k)$

`join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone)` : `C t`

$$= \mu\,(\lambda k.\mathbf{so}\,(\lambda y.k\,(\lambda c.\mathbf{eo}\,(\lambda x.c\,(\mathbf{saw}\,x\,y)))))$$

J, ↑R, ↑L, <

- `someone` : `C e`
  - **Lex**
    - `"someone"`
- `fmap saw everyone` : `C (e -> t)`
  - $\lambda k.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x))$
  - **↑L, >**
    - `saw` : `e -> e -> t`
      - **Lex**
        - `"saw"`
    - `everyone` : `C e`
      - **Lex**
        - `"everyone"`

Higher-order effects: Continuations

- Recall $\mu_{\mathsf{C}} M = \lambda k. M\,(\lambda m. m\,k)$

`join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone) : C t`

$$= \mu\,(\lambda k.\mathbf{so}\,(\lambda y.k\,(\lambda c.\mathbf{eo}\,(\lambda x.c\,(\mathbf{saw}\,x\,y)))))$$

$$= \lambda k.\mathbf{so}\,(\lambda y.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x\,y)))$$

J, ↑R, ↑L, <

`someone : C e`

Lex

`"someone"`

`fmap saw everyone : C (e -> t)`

$\lambda k.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x))$

↑L, >

`saw : e -> e -> t`

Lex

`"saw"`

`everyone : C e`

Lex

`"everyone"`

Higher-order effects: Just an annoying detour

`(<*>) (fmap (\x -> (\f -> f x)) someone) (fmap saw everyone)` : `C t`

$= \lambda k.\,\mathbf{so}\,(\lambda y.\,\mathbf{eo}\,(\lambda x.\,k\,(\mathbf{saw}\,x\,y)))$

A, **<**

`someone` : `C e` `fmap saw everyone` : `C (e -> t)`

`join (fmap (\a -> fmap (\a1 -> a1 a) (fmap saw everyone)) someone)` : `C t`

$= \lambda k.\,\mathbf{so}\,(\lambda y.\,\mathbf{eo}\,(\lambda x.\,k\,(\mathbf{saw}\,x\,y)))$

J, $\uparrow$**R**, $\uparrow$**L**, **<**

`someone` : `C e` `fmap saw everyone` : `C (e -> t)`

- Sad

Higher-order meanings: Continuations

But there is yet another parse:

`fmap (\v -> fmap v someone) (fmap saw everyone) : C (C t)`

↑L, ↑R, <

`someone : C e` — **Lex** — `"someone"`

`fmap saw everyone : C (e -> t)`

$\lambda k.\mathbf{eo}\,(\lambda x.k\,(\mathbf{saw}\,x))$

↑L, >

`saw : e -> e -> t` — **Lex** — `"saw"`

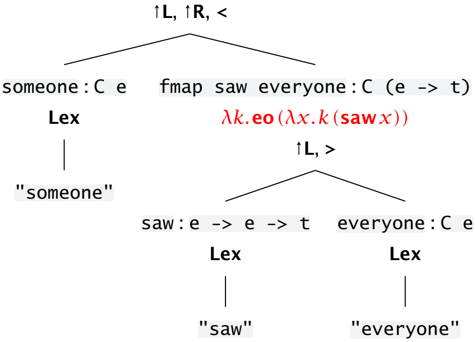
`everyone : C e` — **Lex** — `"everyone"`

Higher-order meanings: Continuations

But there is yet another parse:

`fmap (\v -> fmap v someone) (fmap saw everyone) : C (C t)`

$$(\lambda P.P \bullet \textbf{so}) \bullet (\lambda k.\textbf{eo}(\lambda x.k(\textbf{saw}\,x)))$$



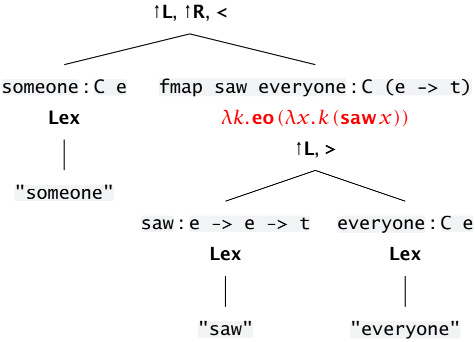
Higher-order meanings: Continuations

But there is yet another parse:

`fmap (\v -> fmap v someone) (fmap saw everyone)` : `C (C t)`

$$(\lambda P.P \bullet \textbf{so}) \bullet (\lambda k.\textbf{eo}(\lambda x.k(\textbf{saw}\,x)))$$

$$\lambda k.\textbf{eo}(\lambda x.k((\textbf{saw}\,x) \bullet \textbf{so}))$$



Higher-order meanings: Continuations

But there is yet another parse:

`fmap (\v -> fmap v someone) (fmap saw everyone)` : `C (C t)`

$(\lambda P.P \bullet \textbf{so}) \bullet (\lambda k.\textbf{eo}(\lambda x.k(\textbf{saw}\, x)))$

$\lambda k.\textbf{eo}(\lambda x.k((\textbf{saw}\, x) \bullet \textbf{so}))$

$\lambda k.\textbf{eo}(\lambda x.k(\lambda c.\textbf{so}(\lambda y.c(\textbf{saw}\, x\, y))))$

↑L, ↑R, <

`someone` : `C e`

Lex

`"someone"`

`fmap saw everyone` : `C (e -> t)`

$\lambda k.\textbf{eo}(\lambda x.k(\textbf{saw}\, x))$

↑L, >

`saw` : `e -> e -> t`

Lex

`"saw"`

`everyone` : `C e`

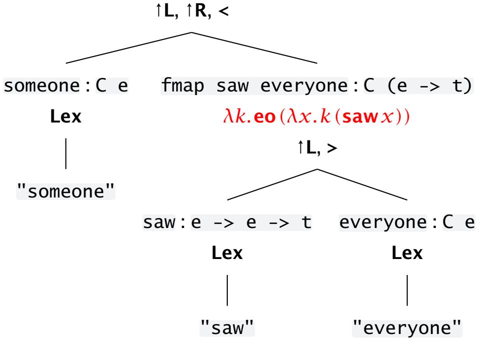
Lex

`"everyone"`

Higher-order meanings: Continuations

- And finally, recalling once more: $\mu_{\mathsf{C}}\, M = \lambda k.\, M\, (\lambda m.\, m\, k)$

`join (fmap (\v -> fmap v someone) (fmap saw everyone))` : `C t`



Higher-order meanings: Continuations

- And finally, recalling once more: $\mu_{\mathsf{C}}\, M = \lambda k.\, M\,(\lambda m.\, m\, k)$

`join (fmap (\v -> fmap v someone) (fmap saw everyone))` : `C t`

$\mu\,(\lambda k.\,\mathbf{eo}\,(\lambda x.\,k\,(\lambda c.\,\mathbf{so}\,(\lambda y.\,c\,(\mathbf{saw}\,x\,y)))))$

↑L, ↑R, <

`someone` : `C e`

Lex

`"someone"`

`fmap saw everyone` : `C (e -> t)`

$\lambda k.\,\mathbf{eo}\,(\lambda x.\,k\,(\mathbf{saw}\,x))$

↑L, >

`saw` : `e -> e -> t`

Lex

`"saw"`

`everyone` : `C e`

Lex

`"everyone"`

Higher-order meanings: Continuations

- And finally, recalling once more: $\mu_C\, M = \lambda k. M\,(\lambda m. m\, k)$

`join (fmap (\v -> fmap v someone) (fmap saw everyone))` : `C t`

$\mu\,(\lambda k.\mathbf{eo}\,(\lambda x. k\,(\lambda c.\mathbf{so}\,(\lambda y. c\,(\mathbf{saw}\,x\,y)))))$

$\lambda k.\mathbf{eo}\,(\lambda x.\mathbf{so}\,(\lambda y. k\,(\mathbf{saw}\,x\,y)))$

↑L, ↑R, <

`someone` : `C e`

Lex

`"someone"`

`fmap saw everyone` : `C (e -> t)`

$\lambda k.\mathbf{eo}\,(\lambda x. k\,(\mathbf{saw}\,x))$

↑L, >

`saw` : `e -> e -> t`

Lex

`"saw"`

`everyone` : `C e`

Lex

`"everyone"`

Higher-order meanings: Not just a detour!

`(<*>) (fmap (\x -> (\f -> f x)) someone) (fmap saw everyone)` : `C t`

$= \lambda k.\mathbf{so}(\lambda y.\mathbf{eo}(\lambda x.k(\mathbf{saw}\,x\,y)))$

A, <

`someone` : `C e`   `fmap saw everyone` : `C (e -> t)`

`join (fmap (\v -> fmap v someone) (fmap saw everyone))` : `C t`

$\lambda k.\mathbf{eo}(\lambda x.\mathbf{so}(\lambda y.k(\mathbf{saw}\,x\,y)))$

J, ↑R, ↑L, <

`someone` : `C e`   `fmap saw everyone` : `C (e -> t)`

- The parser has discovered inverse scope!

Inverse scope via functoriality

The **inverse scope** reading here relies crucially on `fmap`ping one program inside the other

However briefly, you must maintain a moment of higher-order quantification (quantification over programs), unlike the **A**, **<** derivation, where quantification at every step is over simple values

- May be dispreferred relative to regular order (cf. Partee & Rooth 1983)
- Can certainly be distinguished from regular order; beneficial for xover etc[1]

Also it's worth noting that this higher-order route to inverse scope is more powerful than (less parsimoniously) permitting both orders of C's applicative $\circledast$

1. Two people sent a letter to every student. $\forall \gg 2 \gg \exists$

[1] Shan & Barker 2006, Barker & Shan 2008, 2014, Bumford & Charlow 2022

Percolation

With what we have so far, it's easy to see that an effectful type anywhere in a derivation taints everything above it (the effect **percolates upward**)

Particularly eyebrow-raising perhaps is the case of quantificational expressions

$$\texttt{Ct}$$
$$\mathbf{meow} \bullet (\lambda c.\, \neg \exists x.\, \mathbf{cat}\, x \wedge c\, x)$$
$$= \lambda c.\, \neg \exists x.\, \mathbf{cat}\, x \wedge c\, (\mathbf{meow}\, x)$$

$$\texttt{Ce} \qquad\qquad \texttt{Ce} \to \texttt{Ct}$$
$$\lambda c.\, \neg \exists x.\, \mathbf{cat}\, x \wedge c\, x \qquad \lambda E.\, \mathbf{meow} \bullet E$$

no cat

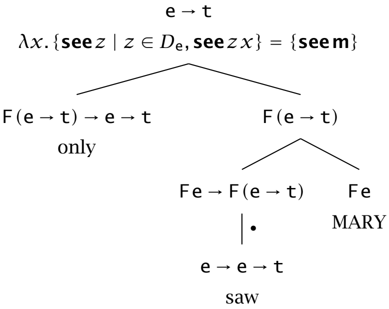
$$\bullet$$
$$\texttt{e} \to \texttt{t}$$
$$\mathbf{meow}$$

meowed

Association with effects

In some cases, there are expressions that **associate with effects**, taking an effectful meaning as argument and returning something pure

Expression	Type	Denotation
only	$\mathsf{F}\,(\mathsf{e} \to \mathsf{t}) \to \mathsf{e} \to \mathsf{t}$	$\lambda\langle P, C\rangle \lambda x.\, \{Q \in C \mid Q\,x\} = \{P\}$



Types ending in $\mathtt{t}$

In other cases, a truth value may be extracted from an effectful meaning in virtue of some broader **linking hypothesis** about how the data structure relates to truth.

These extraction procedures are sometimes called **closure**, or **lowering**, operators, which we might write $\blacksquare_{\mathtt{H}} :: \mathtt{H\,t} \to \mathtt{t}$.

- A sentence with an environmental dependency is true if it is true in the utterance context (cf. Kaplan 1979)
  $\blacksquare_{\mathtt{R}} = \lambda v.\, v\, g_c$
- A sentence with a supplement is true only if both of its dimensions are true (cf. Boër & Lycan 1976)
  $\blacksquare_{\mathtt{W}} = \lambda\langle p, q\rangle.\, p \wedge q$
- A sentence with a presupposition is true only if it is defined and not false (cf. the *A*-ssertion operator of trivalent logics like Beaver & Krahmer 2001)
  $\blacksquare_{\mathtt{M}} = \lambda m.\, \mathtt{false}$ if $m = \#$ else m
- A sentence that evokes many alternatives is true only if one of them is true (cf. Existential Closure, as in Kratzer & Shimoyama 2002)
  $\blacksquare_{\mathtt{S}} = \lambda S.\, \bigvee S$

Closing over continuations

For our scope-taking effect C, the standard closure operator is to run the denotation with a trivial identity continuation (Barker 2002): $\blacksquare_{\mathsf{C}} = \lambda T.\, T\,\mathbf{id}$

$\mathtt{t}$

$\neg\exists x.\,\mathbf{cat}\,x \wedge \mathbf{id}\,(\mathbf{meow}\,x)$

$= \neg\exists x.\,\mathbf{cat}\,x \wedge \mathbf{meow}\,x$

- $\mathtt{C\,t} \to \mathtt{t}$
  $\lambda T.\, T\,\mathbf{id}$
  $\blacksquare_{\mathsf{C}}$
- $\mathtt{C\,t}$
  $\mathbf{meow} \bullet (\lambda c.\, \neg\exists x.\,\mathbf{cat}\,x \wedge c\,x)$
  $= \lambda c.\, \neg\exists x.\,\mathbf{cat}\,x \wedge c\,(\mathbf{meow}\,x)$
  - $\mathtt{C\,e}$
    $\lambda c.\, \neg\exists x.\,\mathbf{cat}\,x \wedge c\,x$
    no cat
  - $\mathtt{C\,e} \to \mathtt{C\,t}$
    $\lambda E.\,\mathbf{meow} \bullet E$
    $\bullet$
    - $\mathtt{e} \to \mathtt{t}$
      $\mathbf{meow}$
      meowed

Obligatory association?

Suppose you have an operator that ‘associates with’/discharges (applicative) effects:

$$\downarrow : F\,a \to a$$

Possible things that might function in this way:

- Pronouns: binders
- Alternatives: ∃-closure
- Focus: focus-sensitive adverbs

Do we predict that association with ↓ will be obligatory?

Non-obligatory association

We do not! Any applicative F allows $\downarrow$ to be ignored:

$$\dfrac{\dfrac{\dfrac{\downarrow : F\,a \to a}{F\,(F\,a \to a)}\,\eta}{F\,(F\,a) \to F\,a}\,\circledast \qquad \dfrac{m : F\,a}{F\,(F\,a)}\,\eta}{F\,a}$$

Beginning with $\{x \mid x \in \textsf{relative}\}$, applying η 'inside' yields $\{\{x\} \mid x \in \textsf{relative}\}$. When $\exists$-closure is folded in, it can target the inner alternatives, sparing the outer.

2. If [∃ a rich relative of mine dies] I'll inherit a house.

You might 'alternatively' take *if* (and *might*) to be alternative-associating in order to capture simplification of disjunctive antecedents (and free choice). See Alonso-Ovalle 2006, Aloni 2007.

The systematicity of very long-distance 'projection'

Functors in general are very useful for percolating effects upward, while leaving the effectful thing in place. And there is a notable tendency of effectful stuff to float up:

3. If [a rich relative of mine dies] I'll inherit a house.
4. Which linguist will be offended if [we invite which philosopher]?
5. [[Dono hon-o yonda] kodomo]-mo yoku nemutta.
which book-ACC read child MO well slept
6. John only gripes when [MARY leaves the lights on].
7. John doesn't gripe when [Mary, a talented linguist, leaves the lights on].
8. John doesn't gripe when [the King of France leaves the lights on].
9. John doesn't gripe when [she leaves the lights on].

Higher-order meanings for selective association

Cross-categorial and higher-order variables (cf. Gardent 1991, Hardt 1993, 1999):

10. …And buy the car she_0 did_1.
11. John_i deposited [his_i paycheck]$_j$. Bill_k spent it_j.
12. Mary_i [likes her_i paper]$_j$. Sam_k does_j too.

Indefinites: potentially selective projection of existential force out of islands.

13. **If** **a persuasive lawyer** visits **a rich relative of mine**, I'll inherit a house.

Focus: potentially selective association of foci with focus-sensitive ops:

14. John only introduced BILL to Mary. He **also only** introduced **BILL** to **SUE**.
15. Last month, John **only** drank BEER. He has **also only** drunk **WINE**.

Charlow 2014, 2019, 2020

Another way down: adjunctions

Nondeterministic state

Dynamic semantics as nondeterministic state: reading, writing, nondeterminism.

$$\begin{aligned} \mathtt{D}\,a &::= \\ \eta\, x &:= \\ m \star f &:= \end{aligned}$$

Nondeterministic state

Dynamic semantics as nondeterministic state: reading, writing, nondeterminism.

$$\begin{aligned} \mathtt{D}\,a &::= \mathtt{s} \to \{a \times \mathtt{s}\} \\ \eta\, x &:= \\ m \star f &:= \end{aligned}$$

Nondeterministic state

Dynamic semantics as nondeterministic state: reading, writing, nondeterminism.

$$\begin{aligned}
\mathtt{D}\,a &::= \mathtt{s} \to \{a \times \mathtt{s}\} \\
\eta\, x &:= \lambda s.\{(x, s)\} \\
m \star f &:=
\end{aligned}$$

Nondeterministic state

Dynamic semantics as nondeterministic state: reading, writing, nondeterminism.

$$\begin{aligned} \mathtt{D}\,a &::= \mathtt{s} \to \{a \times \mathtt{s}\} \\ \eta\, x &\coloneqq \lambda s.\, \{(x, s)\} \\ m \star f &\coloneqq \lambda s.\, \textstyle\bigcup_{(x,s') \in m\, s} f\, x\, s' \end{aligned}$$

Not as simple as it could be

Input, Output, Nondeterminism. Anything that does one does all, even if trivially.

- **she₀** $:= \lambda s.\, \{(s_0, s)\}$
- **mary**$^{+}$ $:= \lambda s.\, \{(\mathsf{m}, s \mathbin{+\!\!+} \mathsf{m})\}$
- **someone** $:= \lambda s.\, \{(x, s) \mid x :: \mathsf{e}\}$

Another sort of generalization to the worst case.

Semantic primitives?

State implicates reading and writing actions (cf. Shan 2001):

$$\mathtt{R}\,a ::= \mathtt{s} \to a \qquad\qquad \mathtt{W}\,a ::= (a, \mathtt{s})$$

$\mathtt{R}$ and $\mathtt{W}$ are **adjoint functors** (in particular, $\mathtt{W} \dashv \mathtt{R}$):

$$F\,a \to b \simeq a \to G\,b$$
$$\mathtt{W}\,a \to b \simeq a \to \mathtt{R}\,b$$
$$(a, \mathtt{s}) \to b \simeq a \to \mathtt{s} \to b$$

W ⊣ R

In fact, R-ing and W-ing are adjoint in virtue of the curry-uncurry isomorphisms:

```
curry :: ((a, s) -> b) -> a -> s -> b   -- (W a -> b) -> a -> R b
curry f a s = f (a, s)

uncurry :: (a -> s -> b) -> (a, s) -> b -- (a -> R b) -> W a -> b
uncurry f (a, s) = f a s

-- curry . uncurry == id
-- uncurry . curry == id
```

Adjunctions in Edward Kmett's Data.Functor.Adjunction

```haskell
class (Functor f, Functor g) => Adjunction f g where
  {-# MINIMAL (unit, counit) | (leftAdjunct, rightAdjunct) #-}
  unit         :: a -> g (f a)
  counit       :: f (g a) -> a
  leftAdjunct  :: (f a -> b) -> a -> g b
  rightAdjunct :: (a -> g b) -> f a -> b

  unit           = leftAdjunct id  -- aka eta
  counit         = rightAdjunct id -- aka epsilon
  leftAdjunct f  = fmap f . unit
  rightAdjunct f = counit . fmap f
```

From adjoints to monads

```haskell
class (Functor f, Functor g) => Adjunction f g where
  {-# MINIMAL (unit, counit) | (leftAdjunct, rightAdjunct) #-}
  unit         :: a -> g (f a) -- eta
  counit       :: f (g a) -> a -- epsilon
  leftAdjunct  :: (f a -> b) -> a -> g b
  rightAdjunct :: (a -> g b) -> f a -> b
```

$F \dashv G$ implies that GF is a monad! We may deduce GF's $\bullet$, η, and μ from $F \dashv G$.

- $\bullet :: (a \to b) \to GF\,a \to GF\,b$ follows from functoriality of G and F
- $\eta :: a \to GF\,a$ is the `unit` of the adjunction
- $\mu :: GFGF\,a \to GF\,a$ is given by $G(\varepsilon)$

From adjoints to monads

```haskell
class (Functor f, Functor g) => Adjunction f g where
  {-# MINIMAL (unit, counit) | (leftAdjunct, rightAdjunct) #-}
  unit         :: a -> g (f a) -- eta
  counit       :: f (g a) -> a -- epsilon
  leftAdjunct  :: (f a -> b) -> a -> g b
  rightAdjunct :: (a -> g b) -> f a -> b
```

$F \dashv G$ implies that GF is a monad! We may deduce GF's $\bullet$, η, and μ from $F \dashv G$.

- $\bullet :: (a \to b) \to GF\,a \to GF\,b$ follows from functoriality of G and F
- $\eta :: a \to GF\,a$ is the `unit` of the adjunction
- $\mu :: GFGF\,a \to GF\,a$ is given by $G(\varepsilon)$

Concretely, for `RW`

```haskell
instance Adjunction W R where
  leftAdjunct  = curry
  rightAdjunct = uncurry
```

Concretely, for RW

```haskell
instance Adjunction W R where
  leftAdjunct  = curry
  rightAdjunct = uncurry

unit x == (leftAdjunct id) x
  == (curry id) x
  == curry (\(a, s) -> (a, s)) x
  == (\a s -> (a, s)) x
  == \s -> (x, s)
```

Concretely, for RW

```haskell
instance Adjunction W R where
  leftAdjunct  = curry
  rightAdjunct = uncurry
```

```haskell
unit x == (leftAdjunct id) x
  == (curry id) x
  == curry (\(a, s) -> (a, s)) x
  == (\a s -> (a, s)) x
  == \s -> (x, s)
```

```haskell
counit (f, x) == (rightAdjunct id) (f, x)
  == (uncurry id) (f, x)
  == (uncurry (\a s -> a s)) (f, x)
  == (\(a, s) -> a s) (f, x)
  == f x
```

Concretely, for RW

```
instance Adjunction W R where
  leftAdjunct  = curry
  rightAdjunct = uncurry
```

```
unit x == (leftAdjunct id) x
  == (curry id) x
  == curry (\(a, s) -> (a, s)) x
  == (\a s -> (a, s)) x
  == \s -> (x, s)
```

```
counit (f, x) == (rightAdjunct id) (f, x)
  == (uncurry id) (f, x)
  == (uncurry (\a s -> a s)) (f, x)
  == (\(a, s) -> a s) (f, x)
  == f x
```

```
join m == fmap_R counit m
  == counit . mm
  == \s -> counit (mm s)
  == \s -> m s' where (m, s') = mm s
```

Monad transformers

What's more, GF can compositionally transform any monad M into a 'super-monad' GMF with the functionality of G, F, and M![2]

- $\bullet :: (a \to b) \to GMF\,a \to GMF\,b$ follows from functoriality of G, M, and F
- $\eta :: a \to GMF\,a$ is given by $G(\eta_M) \circ \eta_{GF}$
- $\mu :: GMFGMF\,a \to GMF\,a$ is given by $G(\mu_M) \circ GM(\varepsilon)$

This is in some sense the "origin" of the State transformer we discussed earlier.

[2] This RL's the `ST` monad, and $R[\ ]L$'s the `ST` **transformer** (Liang, Hudak & Jones 1995, Cohn-Gordon 2016). LR is the `Store` comonad, useful for structured meanings (Krifka 1991, 2006).

Monad transformers

What's more, GF can compositionally transform any monad M into a 'super-monad' GMF with the functionality of G, F, and M![2]

- $\bullet :: (a \to b) \to GMF\,a \to GMF\,b$ follows from functoriality of G, M, and F
- $\eta :: a \to GMF\,a$ is given by $G(\eta_M) \circ \eta_{GF}$
- $\mu :: GMFGMF\,a \to GMF\,a$ is given by $G(\mu_M) \circ GM(\varepsilon)$

This is in some sense the "origin" of the State transformer we discussed earlier.

[2] This RL's the `ST` monad, and $R[\ ]L$'s the `ST` **transformer** (Liang, Hudak & Jones 1995, Cohn-Gordon 2016). LR is the `Store` comonad, useful for structured meanings (Krifka 1991, 2006).

Transformers via adjunctions: Control.Monad.Trans.Adjoint

```haskell
newtype AdjointT f g m a = AdjointT { runAdjointT :: g (m (f a)) }

-- ...
instance (Adjunction f g, Monad m) => Monad (AdjointT f g m) where
  pure = AdjointT . leftAdjunct return
  AdjointT m >>= f =
    AdjointT $ fmap (>>= rightAdjunct (runAdjointT . f)) m
```

Extending type-driven semantic parsing once more

$$\text{if } a \cdot b \Rightarrow (f, c)\text{, then } \begin{cases} F\,a \cdot \,b \Rightarrow (\uparrow\!\mathbf{R}\,f\,l\,r \coloneqq (\lambda l'.\,f\,l'\,r) \bullet l,\ F\,c) \\ \,a \cdot F\,b \Rightarrow (\uparrow\!\mathbf{L}\,f\,l\,r \coloneqq (\lambda r'.\,f\,l\,r') \bullet r,\ F\,c) \\ F\,a \cdot F\,b \Rightarrow (\ \mathbf{A}\ f\,l\,r \coloneqq f \bullet l \circledast r,\ \qquad F\,c) \end{cases}$$

To these binary rules, we can add monadic `join`-ing, and adjoint `counit`:

$$\text{if } a \cdot b \Rightarrow (f, M\,M\,c)\text{, then } a \cdot b \Rightarrow (\mathbf{J}\,f\,l\,r \coloneqq \mu\,(f\,l\,r), M\,c)$$

$$\text{if } a \cdot b \Rightarrow (f, L\,R\,c)\ \text{, then } a \cdot b \Rightarrow (\mathbf{E}\,f\,l\,r \coloneqq \varepsilon\,(f\,l\,r), M\,c)$$

Example: *Mary$_i$'s mom saw her$_i$*

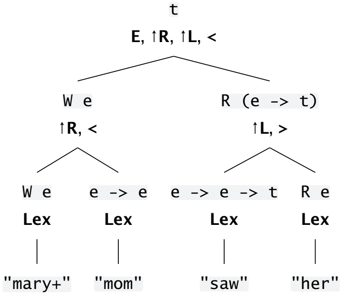
We'll make some very simple (in fact, variable-free) assumptions about meanings:

$$\textbf{mary}^{+} := \underbrace{(\mathsf{m}, \mathsf{m})}_{\mathtt{We}} \qquad \textbf{she} := \underbrace{\lambda x.x}_{\mathtt{Re}}$$

Example: *Mary$_i$'s mom saw her$_i$*

We'll make some very simple (in fact, variable-free) assumptions about meanings:

$$\textbf{mary}^{+} := \underbrace{(\mathsf{m}, \mathsf{m})}_{\texttt{We}} \qquad \textbf{she} := \underbrace{\lambda x . x}_{\texttt{Re}}$$



Equivalent to `saw mary (mom mary)`. Binding without c-command or scope!

Linearity

The picture of binding that emerges from R, W, and their ε, is interestingly **linear** (more precisely, **affine**): every binder can bind (at most) once.

In NL, of course, expressions can have multiple dependents. It would be natural to capture this by allowing pronouns to reactivate their referent in memory:

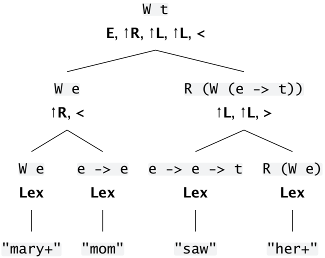
$$\textbf{she}^{+} := \underbrace{\lambda x.\,(x, x)}_{\texttt{R(We)}}$$

Linearity

The picture of binding that emerges from `R`, `W`, and their ε, is interestingly **linear** (more precisely, **affine**): every binder can bind (at most) once.

In NL, of course, expressions can have multiple dependents. It would be natural to capture this by allowing pronouns to reactivate their referent in memory:

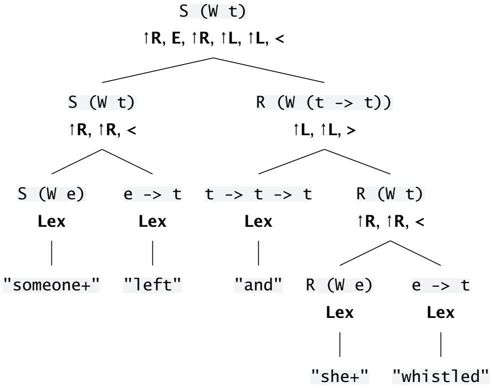
$$\textbf{she}^{+} := \underbrace{\lambda x.\,(x, x)}_{\texttt{R (W e)}}$$



Equivalent to `(saw mary (mom mary), mary)`. The referent lives on!

Someone$_i$ left; and she$_i^+$ whistled (left)

```
*TDParse> semTrees $ parse [someone2, left, and, she2, whistled]
```



Equivalent to `[(left x && whistled x, x) | x <- someone]`!

Comonads: A Higher-Order Detour

Comonads

Remember this very interesting progression from more to less powerful ways that an Effectful computation $\mathtt{F\,a}$ can interact with a continuation k

$$(\bullet) :: (\mathtt{a} \to \mathtt{b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

$$(\circledast) :: \mathtt{F}\,(\mathtt{a} \to \mathtt{b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

$$\mathtt{flip}\,(\star) :: (\mathtt{a} \to \mathtt{F\,b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

There's clearly one other option here...

Comonads

Remember this very interesting progression from more to less powerful ways that an Effectful computation $\mathtt{F\,a}$ can interact with a continuation k

$$(\bullet) :: (\mathtt{a} \to \mathtt{b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

$$(\circledast) :: \mathtt{F}\,(\mathtt{a} \to \mathtt{b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

$$\mathtt{flip}\,(\star) :: (\mathtt{a} \to \mathtt{F\,b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

$$(\dagger) :: (\mathtt{F\,a} \to \mathtt{b}) \to \mathtt{F\,a} \to \mathtt{F\,b}$$

There's clearly one other option here...

Functors with a well-behaved $(\dagger)$ of this type are called **Comonads**

Comonad examples

```haskell
class Comonad f where
  extract :: f a -> a
  extend :: (f a -> b) -> f a -> f b
```

$$\mathsf{F}\,\alpha ::= \ldots\alpha\ldots$$

$$\varepsilon :: \mathsf{F}\,\alpha \to \alpha$$

$$\dagger :: (\mathsf{F}\,\alpha \to \beta) \to \mathsf{F}\,\alpha \to \mathsf{F}\,\beta$$

- Monads often used to model a sequence or pipeline of Effects
- Comonads often used to model interactions with Context

$$\mathsf{W}\,\alpha ::= \langle \alpha, \mathsf{t} \rangle$$

$$\varepsilon = \mathsf{fst}$$

$$k \dagger \langle a, p \rangle = \langle k\,\langle a, p \rangle, p \rangle$$

$$\mathsf{R}\,\alpha ::= \mathsf{r} \to \alpha$$

$$\varepsilon = \lambda w.\, w\,[\,]$$

$$k \dagger w = \lambda r.\, k\,(\lambda r'.\, w\,(r' \mathbin{+\!\!+} r))$$

Comonads from adjunctions

Remember that $F \dashv G$ implies that GF is a monad; it likewise implies that FG is a comonad

```haskell
class (Functor f, Functor g) => Adjunction f g where
  {-# MINIMAL (unit, counit) | (leftAdjunct, rightAdjunct) #-}
  unit         :: a -> g (f a) -- eta
  counit       :: f (g a) -> a -- epsilon
  leftAdjunct  :: (f a -> b) -> a -> g b
  rightAdjunct :: (a -> g b) -> f a -> b
```

We may deduce FG's $\bullet$, ε, and $\dagger$ from $F \dashv G$.

- $\bullet :: (a \to b) \to FG\,a \to FG\,b$ still follows from functoriality of F and G
- $\varepsilon :: FG\,a \to a$ is the `counit` of the adjunction
- $\dagger :: (FG\,a \to b) \to FG\,a \to FG\,b$ is determined from $FG(\eta)$

Comonads from adjunctions

Remember that $F \dashv G$ implies that GF is a monad; it likewise implies that FG is a comonad

```
class (Functor f, Functor g) => Adjunction f g where
  {-# MINIMAL (unit, counit) | (leftAdjunct, rightAdjunct) #-}
  unit         :: a -> g (f a) -- eta
  counit       :: f (g a) -> a -- epsilon
  leftAdjunct  :: (f a -> b) -> a -> g b
  rightAdjunct :: (a -> g b) -> f a -> b
```

We may deduce FG's $\bullet$, ε, and $\dagger$ from $F \dashv G$.

- $\bullet :: (a \to b) \to FG\,a \to FG\,b$ still follows from functoriality of F and G
- $\varepsilon :: FG\,a \to a$ is the `counit` of the adjunction
- $\dagger :: (FG\,a \to b) \to FG\,a \to FG\,b$ is determined from $FG(\eta)$

Concretely, for RW

```haskell
instance Adjunction W R where
  leftAdjunct  = curry
  rightAdjunct = uncurry
```

This means that dual to the RW (State) Monad that you get from the W ⊣ R adjunction, there is also guaranteed to be WR (Costate) Comonad

$$\mathsf{WR}\,\alpha ::= \langle \mathsf{r} \to \alpha, \mathsf{r} \rangle$$

$$\varepsilon \langle c, g \rangle = \langle c\, g, g \rangle$$

$$k \dagger \langle c, g \rangle = \langle \lambda g'. k \langle c, g' \rangle, g \rangle$$

Store comonads for focus

The `WR` structure turns out to be another way to think about focus (Krifka 1992)

Expression	Type	Denotation
SASSY	$\mathtt{F\,e} ::= \mathtt{e} \times \{\mathtt{e}\}$	$\langle \mathbf{s}, \{x \mid x \in D_e\}\rangle$
SASSY sat	$\mathtt{F\,t} ::= \mathtt{t} \times \{\mathtt{t}\}$	$\langle \mathbf{sat}\,\mathbf{s}, \{\mathbf{sat}\,x \mid x \in D_e\}\rangle$
...		
SASSY	$\mathtt{F\,e} ::= (\mathtt{e} \to \mathtt{e}) \times \mathtt{e}$	$\langle \lambda x.x, \mathbf{s}\rangle$
SASSY sat	$\mathtt{F\,t} ::= (\mathtt{e} \to \mathtt{t}) \times \mathtt{e}$	$\langle \lambda x.\mathbf{sat}\,x, \mathbf{s}\rangle$

Extended association with focus

What would it be like to use the † for `WR` considered as a focus effect?

Expression	Type	Denotation
only	$\mathtt{Ft} \to \mathtt{t}$	$\lambda\langle c, x\rangle.\{z \mid c\,z\} = \{x\}$
...		
† only	$\mathtt{Ft} \to \mathtt{Ft}$	$\lambda\langle c, x\rangle.\langle\lambda y.[\![\text{only}]\!]\langle c, y\rangle, x\rangle$
		$= \lambda\langle c, x\rangle.\langle\lambda y.\{z \mid c\,z\} = \{y\}, x\rangle$

Parting words

A lightweight compositional interface that extends familiar compositional semantic theories with effects is within reach.

We can extend type-driven interpretation, simply, with functors, applicatives, monads, adjoints, and possibly other effectful constructs, as the need arises.

We hope to have given you a sense of the power and elegance of this approach, some of the empirical payoffs, and the ways in which it simplifies the task of the semanticist, and perhaps the language learner.

Aloni, Maria. 2007. Free choice, modals, and imperatives. *Natural Language Semantics* 15(1). 65–94. https://doi.org/10.1007/s11050-007-9010-2.

Alonso-Ovalle, Luis. 2006. *Disjunction in alternative semantics*. University of Massachusetts, Amherst Ph.D. thesis. https://semanticsarchive.net/Archive/TVkY2ZlM/.

Barker, Chris. 2002. Continuations and the nature of quantification. *Natural Language Semantics* 10(3). 211–242. https://doi.org/10.1023/A:1022183511876.

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.

Barker, Chris & Chung-chieh Shan. 2014. *Continuations and natural language*. Oxford: Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199575015.001.0001.

Beaver, David & Emiel Krahmer. 2001. A partial account of presupposition projection. *Journal of logic, language and information* 10(2). 147–182.

Boër, Steven E & William G Lycan. 1976. The myth of semantic presupposition.

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

Charlow, Simon. 2019. A modular theory of pronouns and binding. Unpublished ms., Rutgers University. https://ling.auf.net/lingbuzz/003720.

Charlow, Simon. 2020. The scope of alternatives: indefiniteness and islands. *Linguistics and Philosophy* 43(4). 427–472. https://doi.org/10.1007/s10988-019-09278-3.

Cohn-Gordon, Reuben. 2016. Monad transformers for natural language: Combining monads to model effect interaction. Unpublished ms.

Gardent, Claire. 1991. Dynamic semantics and VP-ellipsis. In Jan van Eijck (ed.), *Logics in AI: European workshop JELIA '90 Amsterdam, The Netherlands, September 10–14, 1990 proceedings*, 251–266. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0018446.

Hardt, Daniel. 1993. *VP ellipsis: Form, meaning, and processing*. University of Pennsylvania Ph.D. thesis. https://repository.upenn.edu/dissertations/AAI9331786.

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

Kaplan, David. 1979. On the logic of demonstratives. *Journal of philosophical logic* 8(1). 81–98.

Kratzer, Angelika & Junko Shimoyama. 2002. Indeterminate pronouns: The view from Japanese. In Yukio Otsu (ed.), *Proceedings of the Third Tokyo Conference on Psycholinguistics*, 1–25. Tokyo: Hituzi Syobo.

Krifka, Manfred. 1991. A compositional semantics for multiple focus constructions. In Steve Moore & Adam Wyner (eds.), *Proceedings of Semantics and Linguistic Theory 1*, 127–158. Ithaca, NY: Cornell University. https://doi.org/10.3765/salt.v1i0.2492.

Krifka, Manfred. 1992. A framework for focus-sensitive quantification. In *Semantics and linguistic theory*, vol. 2, 215–236.

Krifka, Manfred. 2006. Association with focus phrases. In Valéria Molnár & Susanne Winkler (eds.), *The Architecture of Focus*, 105–136. Mouton de Gruyter.

Liang, Sheng, Paul Hudak & Mark Jones. 1995. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, 333–343. ACM Press.

Partee, Barbara H. & Mats Rooth. 1983. Generalized conjunction and type ambiguity. In Rainer Bäuerle, Christoph Schwarze & Arnim von Stechow (eds.), *Meaning, Use and Interpretation of Language*, 361–383. Berlin: Walter de Gruyter. `https://doi.org/10.1515/9783110852820.361`.

Shan, Chung-chieh. 2001. A variable-free dynamic semantics. In Robert van Rooy & Martin Stokhof (eds.), *Proceedings of the Thirteenth Amsterdam Colloquium*. University of Amsterdam.

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`.

Wadler, Philip. 1994. Monads and composable continuations. *LISP and Symbolic Computation* 7(1). 39–56. `https://doi.org/10.1007/BF01019944`.