Effectful composition in natural language semantics

From Functors to Applicative Functors

Dylan Bumford (UCLA) Simon Charlow (Rutgers)

ESSLLI 2022, NUI Galway

Type-driven effectful composition

Encoding syntax and semantics

```haskell
data Syn
  = Leaf String
  | Branch Syn Syn

data Sem
  = Lex String
  | Comp Mode Sem Sem
```

```haskell
data Mode
  = FA | BA
  | PM -- etc

data Type
  = E | T
  | Type :-> Type
```

Type-driven combination at the interface

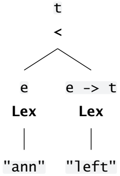
```
combine :: Type -> Type -> [(Mode, Type)]
combine l r =
  [(FA, b) | a :-> b <- [l], a == r] ++
  [(BA, b) | a :-> b <- [r], a == l] ++
  [(PM, a :-> T) | a :-> T <- [l], b :-> T <- [r], a == b]
  -- ...
```

Type-driven combination at the interface

```
combine :: Type -> Type -> [(Mode, Type)]
combine l r =
  [(FA, b) | a :-> b <- [l], a == r] ++
  [(BA, b) | a :-> b <- [r], a == l] ++
  [(PM, a :-> T) | a :-> T <- [l], b :-> T <- [r], a == b]
  -- ...

synsem :: Syn -> [(Sem, Type)]
synsem (Leaf w) = [(Lex w, ty) | ty <- lex w]
synsem (Branch l r) =
  [ (Comp op lval rval, ty) | (lval, lty) <- synsem l
                            , (rval, rty) <- synsem r
                            , (op, ty) <- combine lty rty ]
```

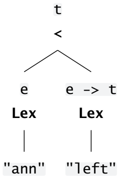
Semantic values as (syntax-homomorphic) trees

```
*TypeDriven> semTrees s0
```

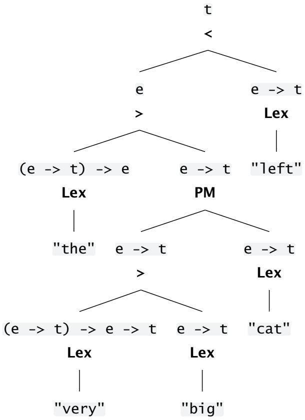


Semantic values as (syntax-homomorphic) trees

```
*TypeDriven> semTrees s0
```




```
*TypeDriven> semTrees s1
```



Adding effectful things to the grammar

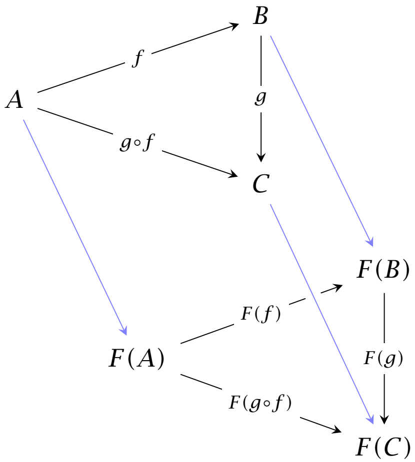
Regular types extended with effect-ful types:

```
data Type = E | T
          | Type :-> Type
          | Eff F Type
```

Some notions of effects to get us going:

```
data F = R
       | S
       | W
       | C
--     ...
```

Then extending our type-driven interpreter just amounts to extending `combine`!



A
B
C
f
g
g∘f
F(A)
F(B)
F(C)
F(f)
F(g)
F(g∘f)

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow \quad c \text{ , then } \begin{cases} F\,a \cdot \quad b \Rightarrow & F\,c \\ \quad a \cdot F\,b \Rightarrow & F\,c \end{cases}$$

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow (f, c)\text{, then} \begin{cases} F\,a \cdot \quad b \Rightarrow & F\,c \\ \quad a \cdot F\,b \Rightarrow & F\,c \end{cases}$$

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow (f, c), \text{ then } \begin{cases} F\,a \cdot \quad b \Rightarrow (\uparrow\!\mathbf{R}\,f\,l\,r := \overbrace{(\lambda l'.\, f\,l'\,r)}^{a \to c} \bullet l, F\,c) \\ \quad a \cdot F\,b \Rightarrow (\uparrow\!\mathbf{L}\,f\,l\,r := \underbrace{(\lambda r'.\, f\,l\,r')}_{b \to c} \bullet r, F\,c) \end{cases}$$

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow (f, c), \text{ then } \begin{cases} F\,a \cdot \quad b \Rightarrow (\uparrow\mathbf{R}\, f\, l\, r := \overbrace{(\lambda l'.\, f\, l'\, r)}^{a \to c} \bullet l, F\,c) \\ \quad a \cdot F\,b \Rightarrow (\uparrow\mathbf{L}\, f\, l\, r := \underbrace{(\lambda r'.\, f\, l\, r')}_{b \to c} \bullet r, F\,c) \end{cases}$$

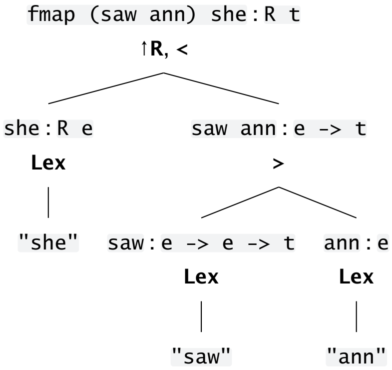
Ported directly to Haskell:

```
combine' :: Type -> Type -> [(Mode, Type)]
combine' l r = combine l r ++
  [ (LR op, Eff f c) | Eff f a <- [l]
                     , functor f, (op, c) <- combine' a r ] ++
  [ (LL op, Eff f c) | Eff f b <- [r]
                     , functor f, (op, c) <- combine' l b ]
```

This technique was developed by Barker & Shan 2014, White et al. 2017 for parsing with continuations. But continuations are functorial, and the technique works for any functor!

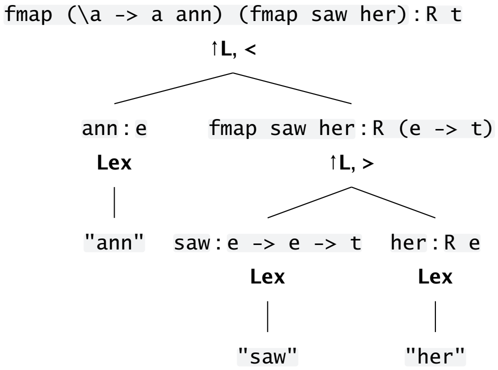
She saw Ann

```
*TDParse> semTrees (parse [she, saw, ann])
```



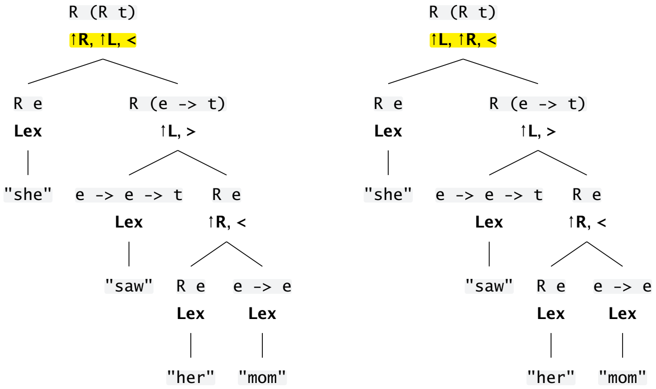
Ann saw her

```
*TDParse> semTrees (parse [ann, saw, her])
```

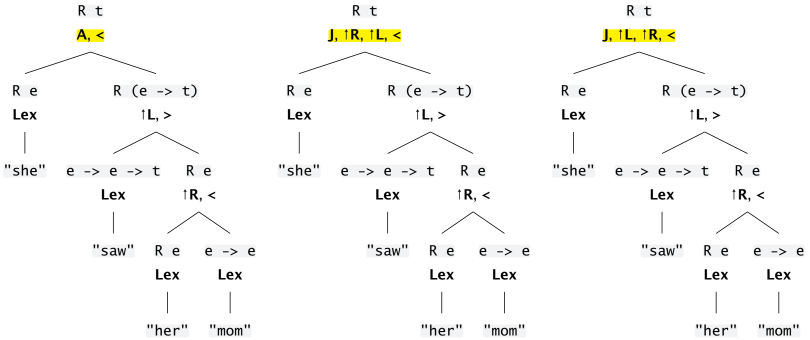


She saw her mom

```
*TDParse> semTrees (parse [she, saw, her, mom])
```



And some mysterious extras…



Some combinations

For any functors F, G, if $a \cdot b \Rightarrow c$, then:

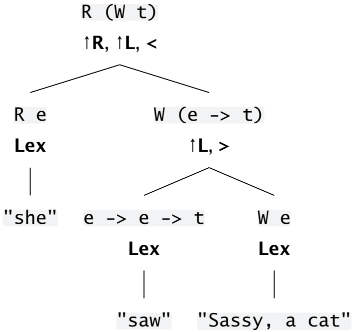
- $a \cdot G\,b \Rightarrow G\,c$
- $F\,a \cdot G\,b \Rightarrow F\,(G\,c)$

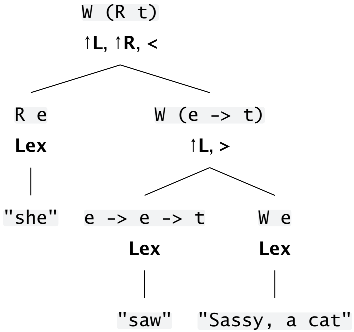
The reverse direction works as well:

- $F\,a \cdot b \Rightarrow F\,c$
- $F\,a \cdot G\,b \Rightarrow G\,(F\,c)$

F and G may be the same, or different.

Pronouns and other effects





Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

How is this practical? LIFT and $(\bullet)$ can apply iteratively.

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

How is this practical? LIFT and $(\bullet)$ can apply iteratively.

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

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

How is this practical? LIFT and $(\bullet)$ can apply iteratively.

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

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

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

How is this practical? LIFT and $(\bullet)$ can apply iteratively.

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

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

- Our semantic parses are **homomorphic** to the syntax that generates them. Construction of derivations is **automatic** (but not complicated for humans).

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

How is this practical? LIFT and $(\bullet)$ can apply iteratively.

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

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

- Our semantic parses are **homomorphic** to the syntax that generates them. Construction of derivations is **automatic** (but not complicated for humans).

What about PM, etc? How does $(\bullet)$ help us with, e.g., *dog near her*?

Comments, questions, concerns addressed

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

- **No, not necessarily**

How is this practical? LIFT and $(\bullet)$ can apply iteratively.

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

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

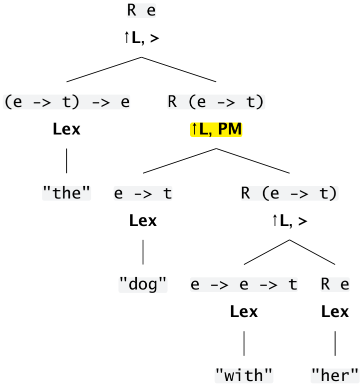
- Our semantic parses are **homomorphic** to the syntax that generates them. Construction of derivations is **automatic** (but not complicated for humans).

What about PM, etc? How does $(\bullet)$ help us with, e.g., *dog near her*?

- Let's check…

It just works!

```
*TDParse> semTrees $ parse [the, dog, with, her]
```



(You could have invented) Applicative Functors!

Environment-dependence

Natural languages have free and bound pro-forms.

1. John saw her. I wouldn't _ if I were you.
2. Everybody$_i$ did their$_i$ homework. When I'm supposed to work$_i$ I can't $\_{}_i$.

It's natural to think of the meanings of these pro-forms as living in a certain Functor representing the effect of depending on (reading from) an environment

$$\sigma ::= \texttt{e} \mid \texttt{t} \mid \sigma \to \sigma \qquad\qquad \tau ::= \texttt{R}\,\sigma ::= \texttt{r} \to \sigma$$

And that composition in the presence of such an effect can be managed by lifting modes of composition **on demand** with `fmap`

The usual story: Heim & Kratzer (1998: 95):

This, however, is not quite the usual story...

(13) *Functional Application* (FA)
If α is a branching node and $\{\beta, \gamma\}$ the set of its daughters, then, for any assignment a, if $[\![\beta]\!]^a$ is a function whose domain contains $[\![\gamma]\!]^a$, then $[\![\alpha]\!]^a = [\![\beta]\!]^a([\![\gamma]\!]^a)$.

(14) *Predicate Modification* (PM)
If α is a branching node and $\{\beta, \gamma\}$ the set of its daughters, then, for any assignment a, if $[\![\beta]\!]^a$ and $[\![\gamma]\!]^a$ are both functions of type $<e,t>$, then $[\![\alpha]\!]^a = \lambda x \in D\ .\ [\![\beta]\!]^a(x) = [\![\gamma]\!]^a(x) = 1$.

In other words, the original argument-structure-driven modes of combination are replaced with counterparts that share environments across constituents

An environmental mode of combination

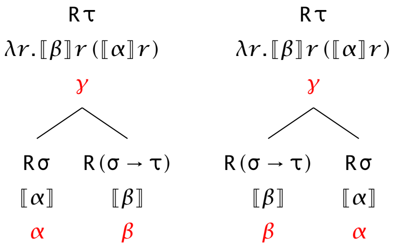
"Function Application"

If a node γ has two daughters

1. α of type $\mathsf{R}\,(\sigma \to \tau)$, and
2. β of type $\mathsf{R}\,\sigma$,

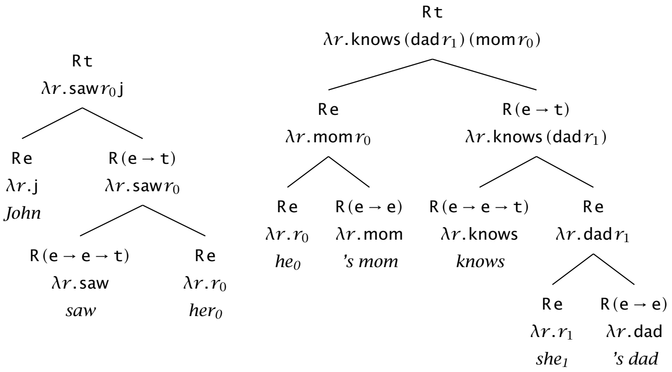
then γ has type $\mathsf{R}\,\tau$, and

$$[\![\gamma]\!] := \underbrace{\lambda r.\,\underbrace{[\![\alpha]\!]\,r}_{\mathsf{R}\,(b\to c)}\,(\underbrace{[\![\beta]\!]\,r}_{\mathsf{R}\,b})}_{\mathsf{R}\,c}$$



In any derivation with **any** pro-form, **every** expression will have to be made environment-sensitive, a kind of **generalization to the worst case**

Environment sharing in action



(Apply the result to a salient environment.)

Pulling out what matters

Key features of the standard approach to environment-dependence:

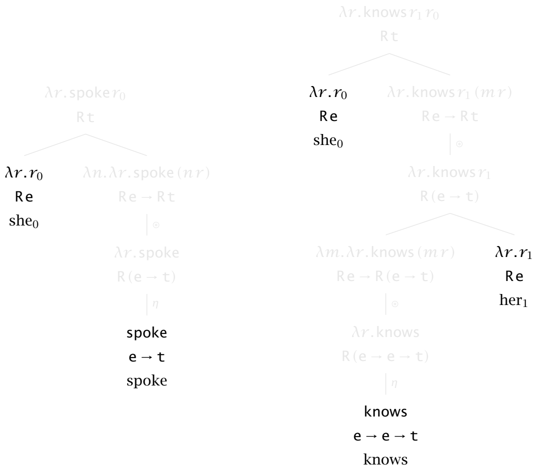
- Uniformity: everything depends on an environment (many things trivially).
- Enriched composition: $[\![\cdot]\!]$ stitches environment-relative meanings together.

Here's another possibility: abstract out these key pieces, apply them on demand.

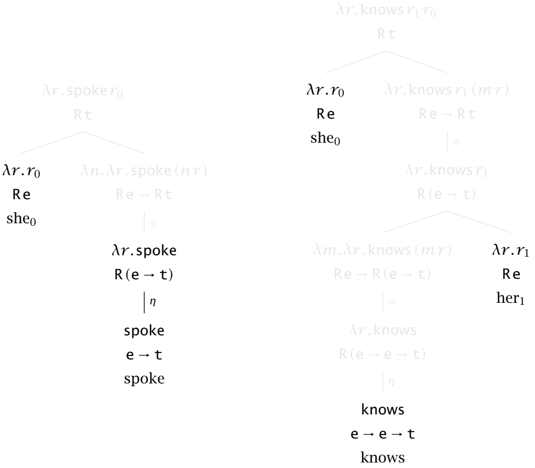
$$\underbrace{\eta\, x := \lambda r.\, x}_{\text{cf. } [\![\text{John}]\!] := \lambda r.\, \mathsf{j}} \qquad \underbrace{m \circledast n := \lambda r.\, m\, r\, (n\, r)}_{\text{cf. } [\![\alpha\ \beta]\!] := \lambda r.\, [\![\alpha]\!]\, r\, ([\![\beta]\!]\, r)}$$

In terms of types, $\eta :: a \to \mathsf{R}\, a$, and $\circledast :: \mathsf{R}\,(a \to b) \to \mathsf{R}\, a \to \mathsf{R}\, b$.

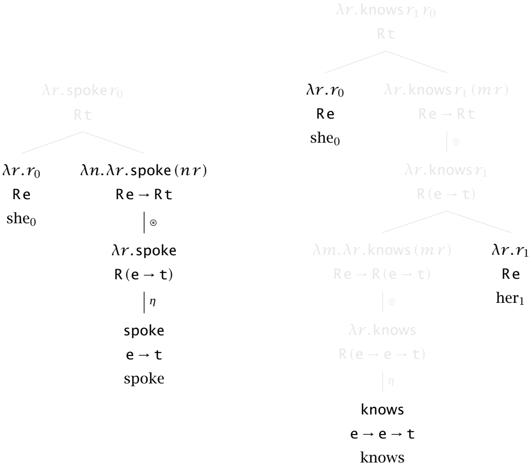
A couple examples



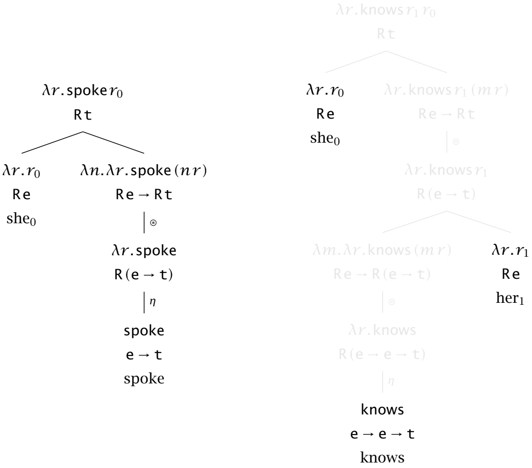
A couple examples



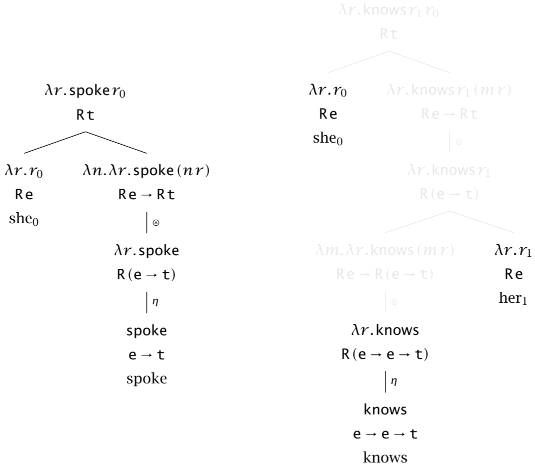
A couple examples



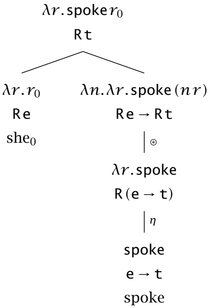
A couple examples

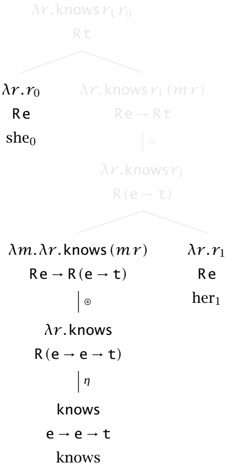


A couple examples

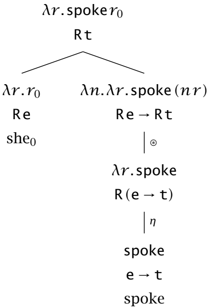


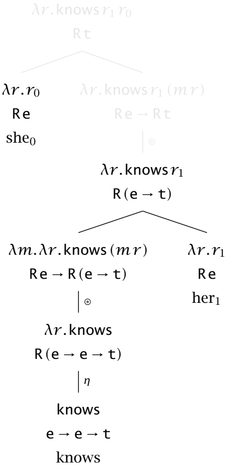
A couple examples



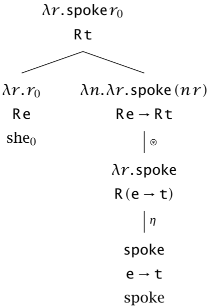


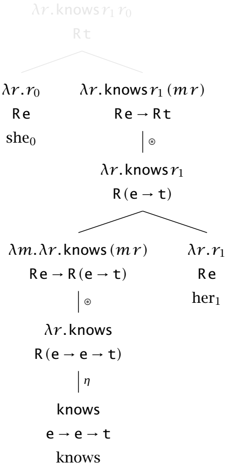
A couple examples



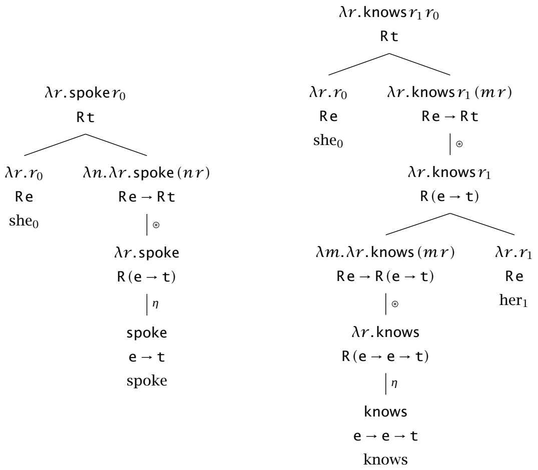


A couple examples





A couple examples



Applicatives

R's η and $\circledast$ make it an **Applicative Functor** (McBride & Paterson 2008, Kiselyov 2015). A type constructor F is applicative if it supports η and $\circledast$ with these types...

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

...Where η is a trivial way to inject something into the richer type characterized by F, and $\circledast$ is function application lifted into F...

Applicatives

R's η and $\circledast$ make it an **Applicative Functor** (McBride & Paterson 2008, Kiselyov 2015). A type constructor F is applicative if it supports η and $\circledast$ with these types...

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

...Where η is a trivial way to inject something into the richer type characterized by F, and $\circledast$ is function application lifted into F...

Homomorphism

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

Identity

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

Interchange

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

Composition

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

Applicatives in Haskell

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The compiler will ensure that the operations you provide are appropriately typed, but it's your job to make sure they're well-behaved.

Nondeterminism[1]

It's common to treat question meanings as sets of possible answers:

3. **Who** ate the ham? $\rightsquigarrow \{\mathsf{ate}\,\mathsf{h}\,x \mid x \in \mathsf{human}\} :: \mathsf{S}\,\mathsf{t}$
4. **Who** ate **what**? $\rightsquigarrow \{\mathsf{ate}\,y\,x \mid x \in \mathsf{human}, y \in \mathsf{thing}\} :: \mathsf{S}\,\mathsf{t}$

Naturally handled using another applicative functor, for sets::

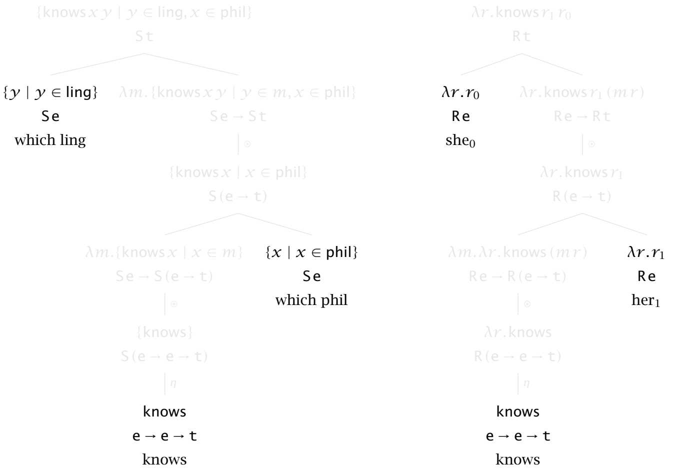
$$\underbrace{\eta\,x := \{x\}}_{\eta :: a \to \mathsf{S}\,a} \qquad \underbrace{m \circledast n := \{f\,x \mid f \in m, x \in n\}}_{\circledast :: \mathsf{S}\,(a \to b) \to \mathsf{S}\,a \to \mathsf{S}\,b}$$

Nondeterministic meanings also evident in:

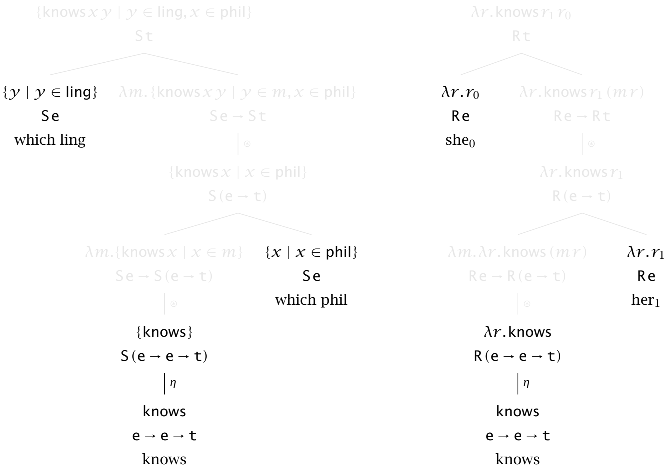
5. You may eat an apple **or** a pear. $\vDash$ You may eat an apple.
   Mail the letter. $\nvDash$ Mail **or** burn the letter.
6. Take **a card**. Place **it** on the bottom of the deck.

[1] Cf. Hamblin 1973, Shan 2001, Charlow 2014, 2020.

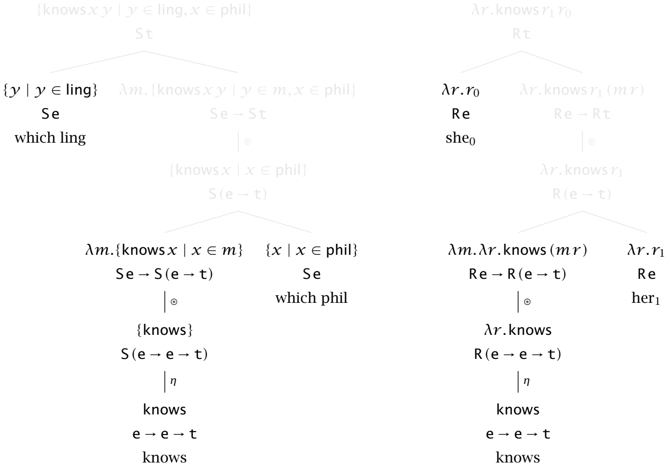
Sample derivation, compared with environment-sensitivity



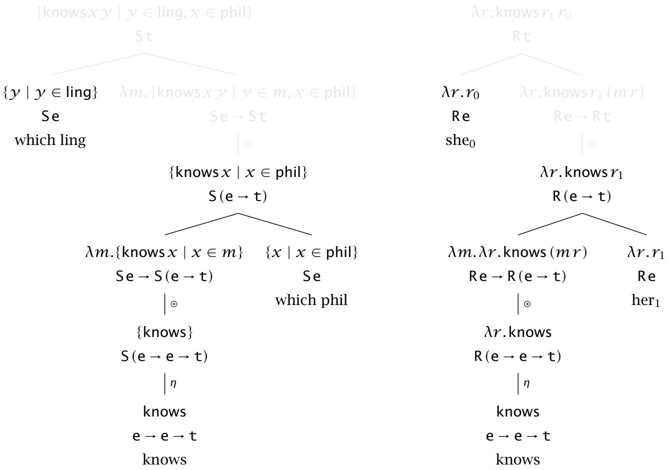
Sample derivation, compared with environment-sensitivity



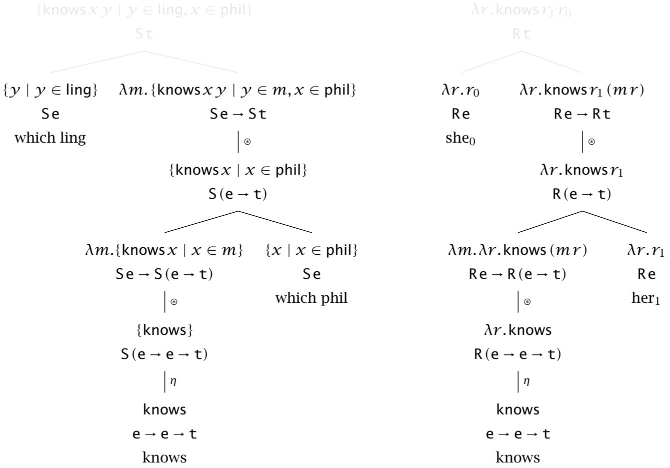
Sample derivation, compared with environment-sensitivity



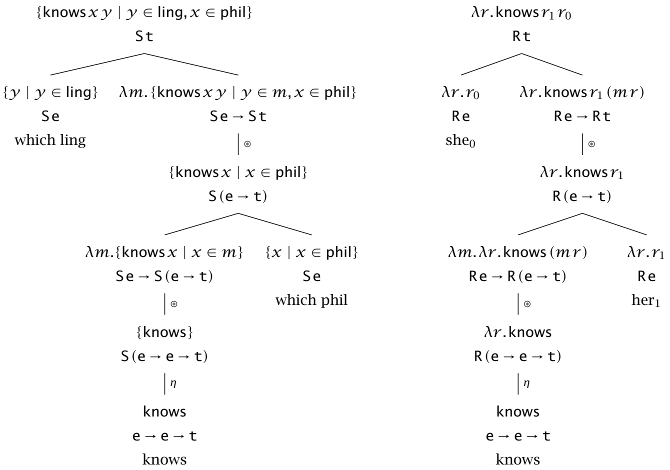
Sample derivation, compared with environment-sensitivity



Sample derivation, compared with environment-sensitivity



Sample derivation, compared with environment-sensitivity



Supplementation[2]

Some expressions contribute information in a secondary "not-at-issue" register:

7. Joe, a linguist, lectured. $\rightsquigarrow$ $(\mathtt{lectured\,j}, [\mathtt{ling\,j}]) :: \mathtt{W\,t}$
8. Joe, a linguist, knows Mary, a philosopher. $\rightsquigarrow$ $(\mathtt{knows\,m\,j}, [\mathtt{ling\,j}, \mathtt{phil\,m}]) :: \mathtt{W\,t}$
9. Polly hasn't read W&P, which is a classic. $\rightsquigarrow$ $(\neg\mathtt{read\,w\&p\,p}, [\mathtt{classic\,w\&p}]) :: \mathtt{W\,t}$

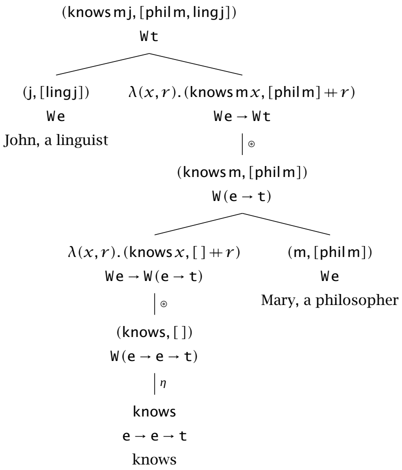
Another example of an applicative functor, for supplements:

$$\underbrace{\eta\, x := (x, [\,])}_{\eta :: a \to \mathtt{W}\, a} \qquad \underbrace{(f, l) \circledast (x, r) := (f\, x, l \mathbin{+\!\!+} r)}_{\circledast :: \mathtt{W}\,(a \to b) \to \mathtt{W}\, a \to \mathtt{W}\, b}$$

In fact, pairs are applicative whenever the second element is monoidal. Why?

[2] Cf. Potts (2005), Giorgolo & Asudeh (2012), and AnderBois, Brasoveanu & Henderson (2015).

Sample derivation: Supplementation



Intonational focus

Contrastive focus invokes alternatives to what was said:

10. I only introduced {Jennifer, JENNIFER} to {Bill, BILL}.
11. Who did you introduce Jennifer to?
    I introduced Jennifer (not JENNIFER) to BILL (not Bill).

Here, $\mathsf{F}\,a ::= a \times \mathsf{S}\,a$, with the following applicative operations (Rooth 1985):

$$\eta\, x := (x, \{x\}) \qquad (f, S) \circledast (x, T) := (f\,x, \{s\,t \mid s \in S, t \in T\})$$

Using this applicative, we can derive the following meanings:

12. I introduced JENNIFER to Bill. $\rightsquigarrow \{\mathsf{intro}\, x\, \mathsf{b}\, \mathsf{i} \mid x \in \mathsf{alt}_\mathsf{j}\}$
13. I introduced Jennifer to BILL. $\rightsquigarrow \{\mathsf{intro}\, \mathsf{j}\, y\, \mathsf{i} \mid y \in \mathsf{alt}_\mathsf{b}\}$
14. I introduced JENNIFER to BILL. $\rightsquigarrow \{\mathsf{intro}\, x\, y\, \mathsf{i} \mid x \in \mathsf{alt}_\mathsf{j}, y \in \mathsf{alt}_\mathsf{b}\}$

Scope and continuations

Languages have quantificational expressions, and they take scope:

15. Every lecturer presented in a room on the third floor.
$\rightsquigarrow \forall\,(\lambda x.\,\exists\,(\lambda y.\,\mathsf{pres}\,y\,x))$
$\rightsquigarrow \exists\,(\lambda y.\,\forall\,(\lambda x.\,\mathsf{pres}\,y\,x))$

The relevant enrichment handles expressions with a scope (continuation):[3]

$$\mathsf{C}\,a ::= (a \to \mathsf{t}) \to \mathsf{t} \qquad \forall, \exists :: \mathsf{C}\,\mathsf{e} = (\mathsf{e} \to \mathsf{t}) \to \mathsf{t}$$

Yet another example of an applicative functor, for scope (continuations):

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

[3] Shan (2001), Barker (2002), Shan & Barker (2006), Barker & Shan (2008, 2014), and Charlow (2014).

Sample derivation: Scope

$\lambda k.\forall x.k\,(\mathtt{spoke}\,x)$
$\mathtt{C\,t}$

$\lambda k.\forall x.k\,x$
$\mathtt{C\,e}$
everyone

$\lambda n.\lambda k.n\,(\lambda x.k\,\mathtt{spoke}\,x)$
$\mathtt{C\,e} \to \mathtt{C\,t}$

$\circledast$

$\lambda k.k\,\mathtt{spoke}$
$\mathtt{C\,(e \to t)}$

η

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

Sample derivation: Scope

$\lambda k.\forall x.k\,(\mathsf{spoke}\,x)$

$\mathsf{C\,t}$

$\lambda k.\forall x.k\,x$	$\lambda n.\lambda k.n\,(\lambda x.k\,\mathsf{spoke}\,x)$
$\mathsf{C\,e}$	$\mathsf{C\,e} \to \mathsf{C\,t}$
everyone	$\circledast$
	$\lambda k.k\,\mathsf{spoke}$
	$\mathsf{C\,(e \to t)}$
	η
	spoke
	$\mathsf{e \to t}$
	spoke

Sample derivation: Scope

$\lambda k.\forall x.k\,(\mathsf{spoke}\,x)$

$\mathsf{C\,t}$

$\lambda k.\forall x.k\,x$

$\mathsf{C\,e}$

everyone

$\lambda n.\lambda k.n\,(\lambda x.k\,\mathsf{spoke}\,x)$

$\mathsf{C\,e} \to \mathsf{C\,t}$

$\circledast$

$\lambda k.k\,\mathsf{spoke}$

$\mathsf{C\,(e \to t)}$

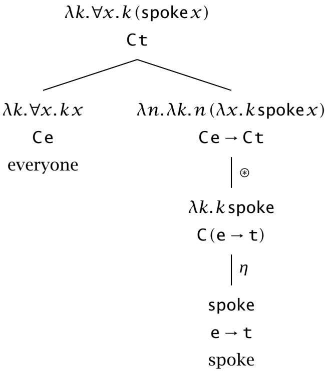
η

spoke

$\mathsf{e \to t}$

spoke

Sample derivation: Scope



Scope alternations via flexibility in $\circledast$

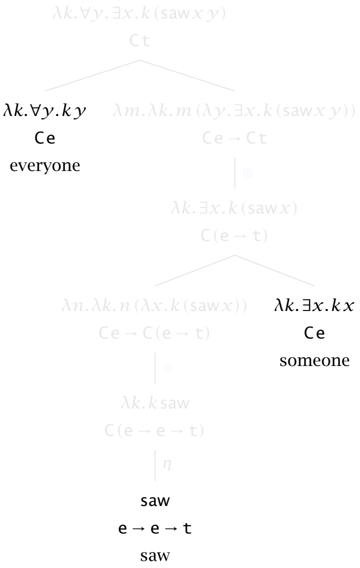
The Continuations applicative is non-commutative in that it admits two $\circledast$'s which evaluate their arguments in opposite orders.

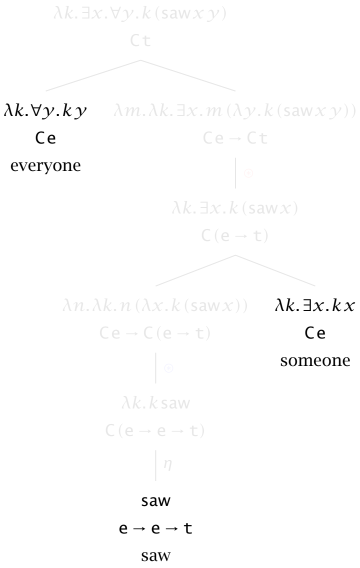
$$\eta\, x := \lambda k.\, k\, x$$

$$\underbrace{m \circledast n := \lambda k.\, m\, (\lambda f.\, n\, (\lambda x.\, k\, (f\, x)))}_{\text{function-first}}$$

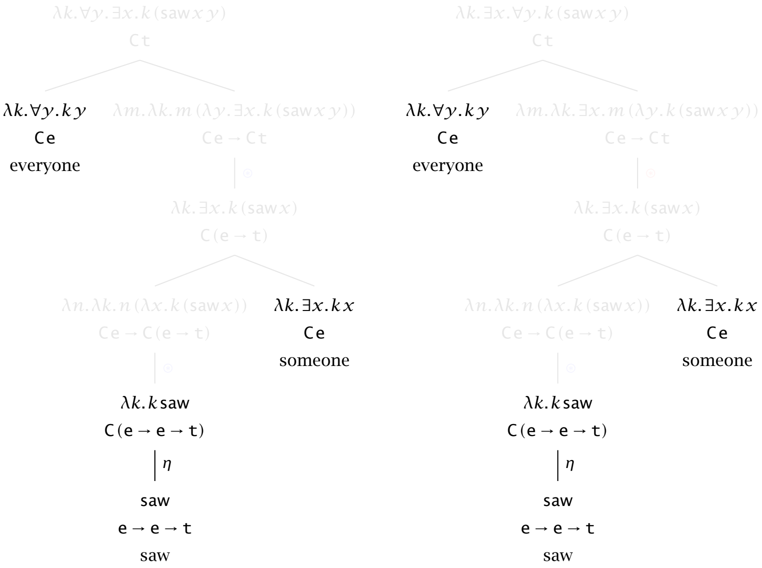
$$\underbrace{m \circledast n := \lambda k.\, n\, (\lambda x.\, m\, (\lambda f.\, k\, (f\, x)))}_{\text{argument-first}}$$

A couple examples

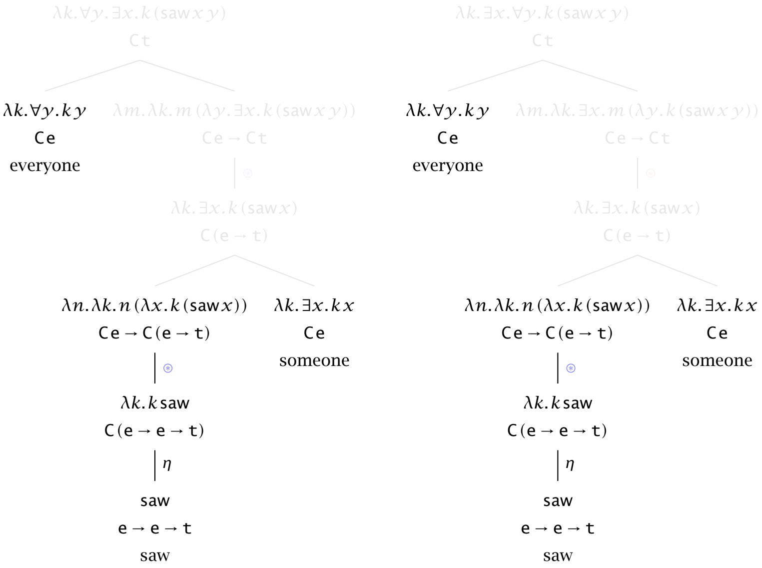




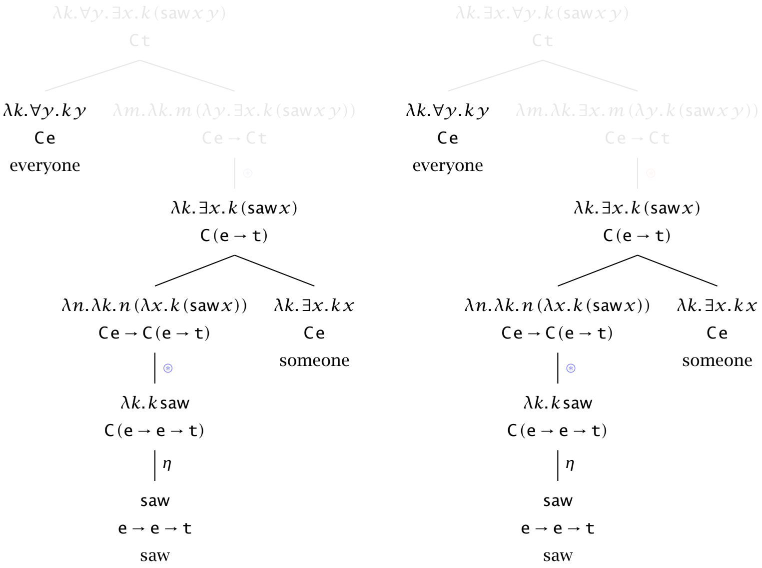
A couple examples



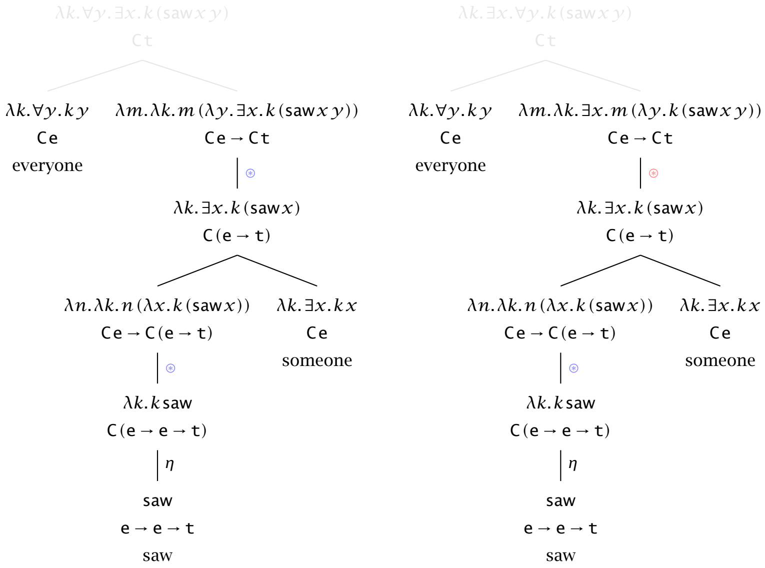
A couple examples



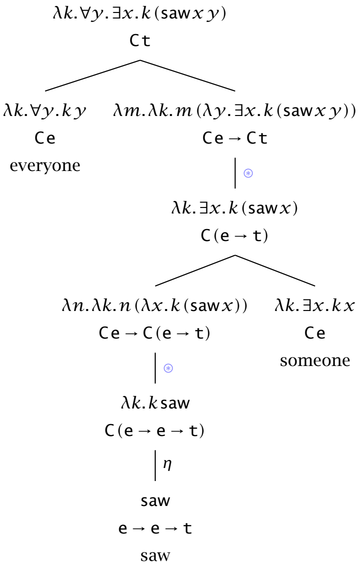
A couple examples

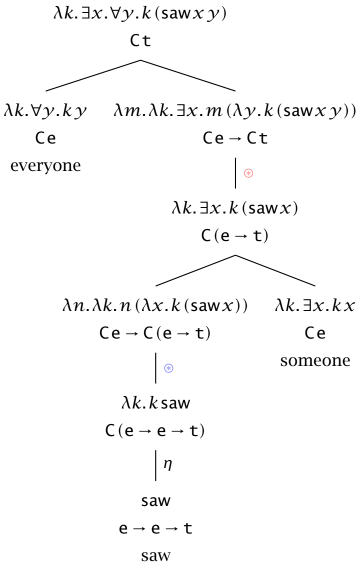


A couple examples



A couple examples





Corresponding notions in programming

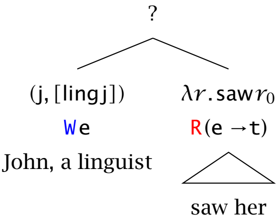
- Pronouns and pronominal binding
- Questions/‘inquisitive’ meanings
- Focus
- Presupposition
- Supplemental content
- Quantification

- Variable management
- Nondeterministic computation
- Cellular automata
- Throwing and catching errors
- Logging/execution traces
- Control flow (jumps, aborts, loops)

Reading and `Writing`: A case study in composition

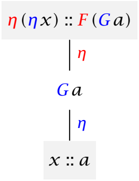
Simultaneous applicative effects

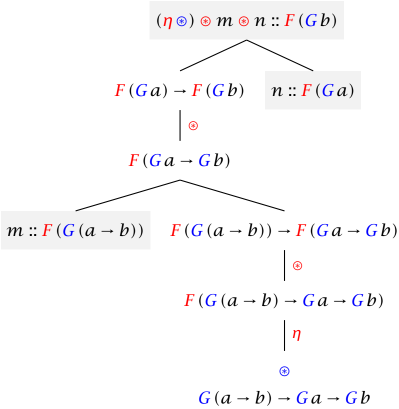
How to combine expressions from different *applicative* effect regimes?



Let's not hand-roll new modes of combination for every combination of effects!

Applicative functors compose, too!





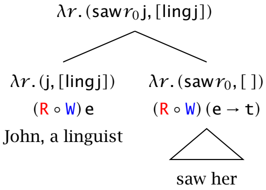
Ross Paterson's Data.Functor.Compose (on Hackage)

```haskell
module Data.Functor.Compose (
    Compose(..),
   ) where

newtype Compose f g a = Compose { getCompose :: f (g a) }

instance (Applicative f, Applicative g) =>
    Applicative (Compose f g) where
      pure x = Compose (pure (pure x))
      Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

Composition with composition

Here's what we get for the composition of R and W, $(\mathtt{R} \circ \mathtt{W})\, a = \mathtt{r} \to (a, [\mathtt{t}])$:

$$\eta\, x \coloneqq \lambda r.\, (x, [\,]) \qquad m \circledast n \coloneqq \lambda r.\, (f\, x, j +\!\!+ k) \textbf{ where } (f, j) \coloneqq m\, r$$
$$(x, k) \coloneqq n\, r$$



$\mathtt{R} \circ \mathtt{W}$ also implies ways to lift $\mathtt{R}\, a$ and $\mathtt{W}\, a$ into $(\mathtt{R} \circ \mathtt{W})\, a$. Exercise: find them!

Some more composed applicatives[4]

Whenever F and G are applicative, $F \circ G$ is too. Here, for $\mathsf{R} \circ \mathsf{S}$:

$$\eta\, x := \lambda r.\{x\} \qquad m \circledast n := \lambda r.\{f\,x \mid f \in m\,r, x \in n\,r\}$$
$$= \eta\,(\eta\,x) \qquad\qquad = (\eta \circledast) \circledast m \circledast n$$

And here, for $\mathsf{S} \circ \mathsf{R}$:

$$\eta\, x := \{\lambda r.x\} \qquad m \circledast n := \{\lambda r.f\,r\,(x\,r) \mid f \in m, x \in n\}$$
$$= \eta\,(\eta\,x) \qquad\qquad = (\eta \circledast) \circledast m \circledast n$$

[4] Cf. Rooth (1985), Kratzer & Shimoyama (2002), Romero & Novel (2013), and Charlow (2020).

Reading what's been written

You might think that with the capacity to both push and pull things from a context, we ought to be able to capture some kinds of anaphora.

16. $\underbrace{\text{Polly}}_{\text{Write}}$ walked in the park. $\underbrace{\text{She}}_{\text{Read}}$ whistled.

Composing reading and writing actions

The reader/writer composition, with an entity-log:

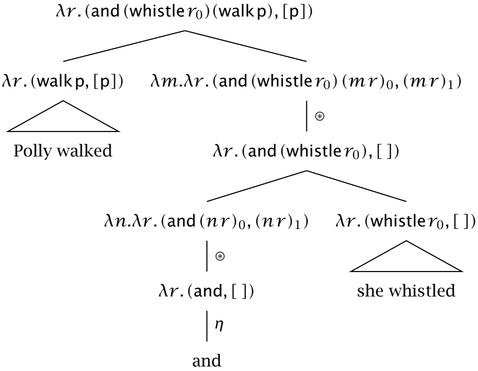
$$(\mathtt{R} \circ \mathtt{W})\, a ::= \mathtt{r} \to (a, [\mathtt{e}])$$

And the corresponding η and $\circledast$ operations again:

$$\eta x := \lambda r.\, (x, [\,]) \qquad m \circledast n := \lambda r.\, (f\,x, j + k) \textbf{ where } \begin{aligned}(f, j) &:= m\,r \\ (x, k) &:= n\,r\end{aligned}$$

Not quite what we're after: the modified state output by m is not passed in to n.

Failure to communicate



The pronoun Reads and the proper name Writes, but they don't coordinate.

Another method of effect composition

But this nevertheless seems like the right structure to manage this sort of effect, and in fact, there is a second applicative for this type.

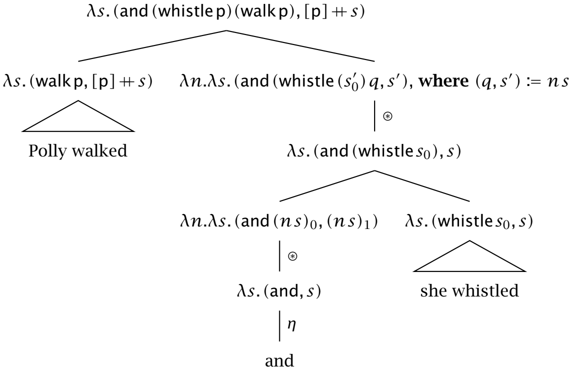
The State applicative: $\mathsf{ST}\, a ::= \mathsf{s} \to (a, \mathsf{s})$

$$\eta\, x := \lambda s.\, (x, s) \qquad m \circledast n := \lambda s.\, (f\, x, s'') \textbf{ where } \begin{aligned}(f, s') &= m\, s \\ (x, s'') &= n\, s'\end{aligned}$$

$$\eta\, x = \lambda r.\, (x, [\,]) \qquad m \circledast n = \lambda r.\, (f\, x, j + k) \textbf{ where } \begin{aligned}(f, j) &:= m\, r \\ (x, k) &:= n\, r\end{aligned}$$

Crucially, the modified state s' is passed into n.

Successful communication

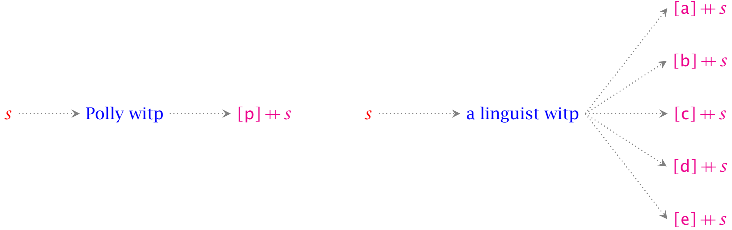


The proper name Writes something the pronoun Reads.

Indefinites, interleaving another effect

Indefinites combine reading and writing with nondeterminism:[5]

17. Polly walked in the park. She whistled.
18. A linguist walked in the park. She whistled.



The nondeterministic state applicative, $\mathsf{D}a ::= \mathsf{s} \rightarrow \mathsf{S}\,(a \times s)$:

$$\eta\, x := \lambda s.\{(x, s)\} \qquad m \circledast n := \lambda s.\{(f\,x, s'') \mid (f, s') \in m\,s, (x, s'') \in n\,s'\}$$

[5] Heim (1982), Barwise (1987), Rooth (1987), Groenendijk & Stokhof (1991), and Muskens (1996), etc.

Semantic parsing with applicatives

There is almost nothing more to say

Extending `combine` with **applicatives**

$$\text{if } a \cdot b \Rightarrow c \text{, then } \begin{cases} F\,a \cdot b \Rightarrow F\,c \\ a \cdot F\,b \Rightarrow F\,c \\ F\,a \cdot F\,b \Rightarrow F\,c \end{cases}$$

Extending `combine` with **applicatives**

$$\text{if } a \cdot b \Rightarrow (f, c)\text{, then } \begin{cases} F\,a \cdot \quad b \Rightarrow & F\,c \\ \quad a \cdot F\,b \Rightarrow & F\,c \\ F\,a \cdot F\,b \Rightarrow & F\,c \end{cases}$$

Extending `combine` with **applicatives**

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

Extending `combine` with **applicatives**

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

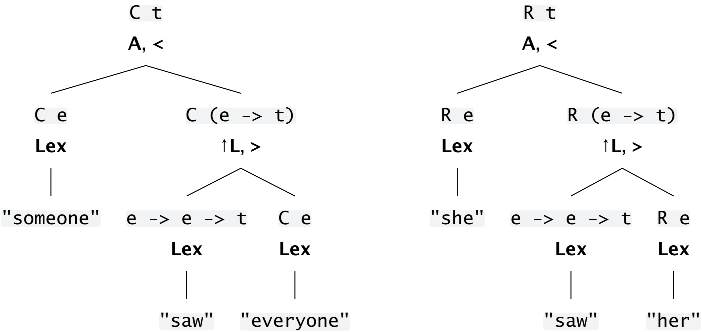
Extending `combine` with **applicatives**

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

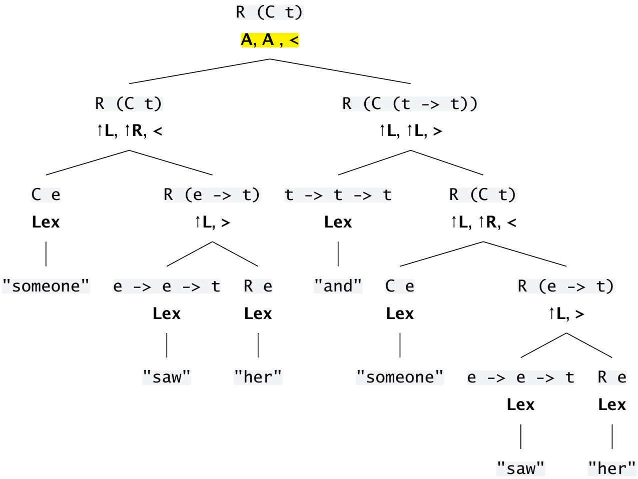
Ported directly to Haskell, again:

```
combine' :: Type -> Type -> [(Mode, Type)]
combine' l r = combine l r ++
  [ (LR op, Eff f c) | Eff f a <- [l]
                     , functor f, (op, c) <- combine' a r ] ++
  [ (LL op, Eff f c) | Eff f b <- [r]
                     , functor f, (op, c) <- combine' l b ] ++
  [ (A  op, Eff f c) | Eff f a <- [l], Eff g b <- [r], f == g
                     , applicative f, (op, c) <- combine' l b ]
```

Deriving regular-order meanings using **A**



Composition of applicatives



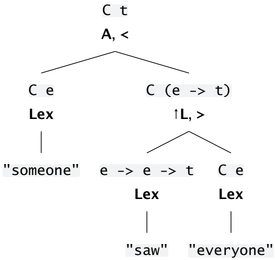
A few words on continuations

Two types of **A**'s entertained earlier for C: function- or argument-first. What happens here?

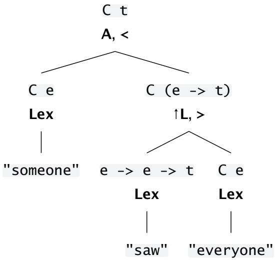
- If $a \cdot b \Rightarrow (f, c)$, $F\,a \cdot F\,b \Rightarrow (\mathbf{A}\,f\,l\,r \coloneqq f \bullet l \circledast r)$

For C (with function-first $\circledast$) this gives $l \gg r$. There's systematic linear bias in composition!

- $\mathbf{A}\,f\,l\,r \underset{\mathsf{C}}{\rightsquigarrow} \lambda k.\, l\,(\lambda l'.\, r\,(\lambda r'.\, f\,l'\,r'))$



A few words on continuations



Two types of **A**'s entertained earlier for C: function- or argument-first. What happens here?

- If $a \cdot b \Rightarrow (f, c)$, $F\,a \cdot F\,b \Rightarrow (\mathbf{A}\,f\,l\,r := f \bullet l \circledast r)$

For C (with function-first $\circledast$) this gives $l \gg r$. There's systematic linear bias in composition!

- $\mathbf{A}\,f\,l\,r \underset{\mathsf{C}}{\rightsquigarrow} \lambda k.l\,(\lambda l'.r\,(\lambda r'.f\,l'\,r'))$

What about inverse scope? It arises in higher-order (functorial) derivations. These:

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

Shan & Barker 2006, Barker & Shan 2008, 2014, Bumford & Charlow 2022

Future work

Ultimately, we are hand-rolling much of what the Haskell compiler already does so well (and in a less type-safe way). It would be preferable to not re-invent the wheel.

There are inefficiencies in the naive version of applicative parsing sketched here. Partially remedied w/a notion of normal form derivations (White et al. 2017).

- Neural parsing achieves state-of-the-art accuracy and speed without dynamic programming (Lee, Lewis & Zettlemoyer 2016). Something else to try.

AnderBois, Scott, Adrian Brasoveanu & Robert Henderson. 2015. At-issue proposals and appositive impositions in discourse. *Journal of Semantics* 32(1). 93–138. https://doi.org/10.1093/jos/fft014.

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.

Barwise, Jon. 1987. Noun phrases, generalized quantifiers, and anaphora. In Peter Gärdenfors (ed.), *Generalized Quantifiers*, 1–29. Dordrecht: Reidel. https://doi.org/10.1007/978-94-009-3381-1_1.

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

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.

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.

Groenendijk, Jeroen & Martin Stokhof. 1991. Dynamic predicate logic. *Linguistics and Philosophy* 14(1). 39–100. https://doi.org/10.1007/BF00628304.

Hamblin, C. L. 1973. Questions in Montague English. *Foundations of Language* 10(1). 41–53.

Heim, Irene. 1982. *The semantics of definite and indefinite noun phrases*. University of Massachusetts, Amherst Ph.D. thesis. https://semanticsarchive.net/Archive/Tk0ZmYyY/.

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

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.

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.

Lee, Kenton, Mike Lewis & Luke Zettlemoyer. 2016. Global neural CCG parsing with optimality guarantees. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2366–2376. Austin, Texas: Association for Computational Linguistics. https://doi.org/10.18653/v1/D16-1262. https://aclanthology.org/D16-1262.

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

Muskens, Reinhard. 1996. Combining Montague semantics and discourse representation. *Linguistics and Philosophy* 19(2). 143–186. https://doi.org/10.1007/BF00635836.

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.

Potts, Christopher. 2005. *The logic of conventional implicatures*. Oxford: Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199273829.001.0001.

Romero, Maribel & Marc Novel. 2013. Variable binding and sets of alternatives. In Anamaria Fălăuș (ed.), *Alternatives in Semantics*, chap. 7, 174–208. London: Palgrave Macmillan UK. https://doi.org/10.1057/9781137317247_7.

Rooth, Mats. 1985. *Association with focus*. University of Massachusetts, Amherst Ph.D. thesis.

Rooth, Mats. 1987. Noun phrase interpretation in Montague grammar, File Change Semantics, and situation semantics. In Peter Gärdenfors (ed.), *Generalized Quantifiers*, 237–269. Dordrecht: Reidel. https://doi.org/10.1007/978-94-009-3381-1_9.

Shan, Chung-chieh. 2001. Monads for natural language semantics. In Kristina Striegnitz (ed.), *Proceedings of the ESSLLI 2001 Student Session*, 285–298. https://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.

White, Michael, Simon Charlow, Jordan Needle & Dylan Bumford. 2017. Parsing with dynamic continuized CCG. In *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+13)*, 71–83. Umeå, Sweden: Association for Computational Linguistics. http://aclweb.org/anthology/W17-6208.