What is it to know a language?

Syntax: which structures are in $\mathcal{L}$, and how they're built.

Semantics: how structures in $\mathcal{L}$ are systematically associated with meaning.

Semantics is difficult, in part, because meaning is both multi- and high-dimensional:

- environment dependence
- nondeterministic
- at-issue and not-at-issue,
- contrastive
- stateful
- quantificational/scopal, ...

Today:

- motivate analogies between semantics and functional programming
- introduce **Functors** to model dimensions of meaning resembling **side effects** in programming

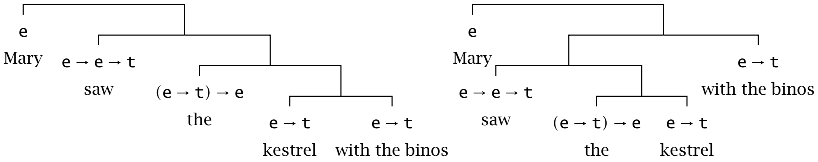
Two ways syntax matters

Only some strings of words are recognizably part of (e.g.) English:

1. Matt devoured the donut.
2. *Matt donut the devoured.
3. *Matt devoured the donut Mary.

And some strings can be understood in multiple ways:

4. Mary saw the kestrel with the binoculars.



We'll assume that this much syntax is provided for us

An arithmetic language and evaluator in Haskell

```haskell
-- Syntax: wffs are those that typecheck as Term's
data Term = Lit Int | Term :+: Term | Term :*: Term

exp1 :: Term
exp1 = Lit 1 :+: (Lit 2 :*: Lit 3)

exp2 :: Term
exp2 = (Lit 1 :+: Lit 2) :*: Lit 3

-- Semantics: (recursively) evaluating terms
eval :: Term -> Int
eval (Lit x)   = x
eval (a :+: b) = (eval a) + (eval b)
eval (a :*: b) = (eval a) * (eval b)
  -- eval exp1 = 7
  -- eval exp2 = 9
```

Types and (higher-order) functions

If you ask the Haskell interpreter about the **types** of the addition operations:

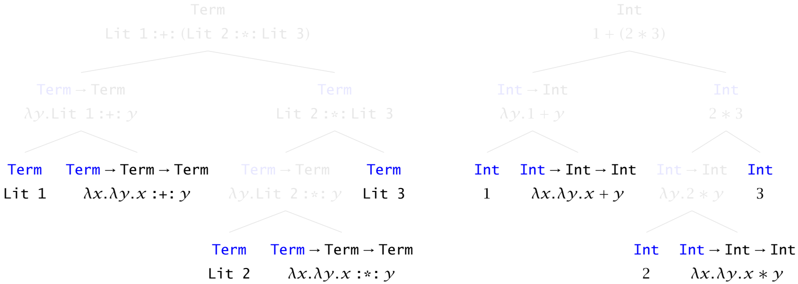
```
GHCi> :type (+)
(+) :: Int -> Int -> Int

GHCi> :type (:+:)
(:+:) :: Term -> Term -> Term
```

This says that (+) is needs one `Int`, and then another, in order to produce an `Int`

Likewise, the term constructor `:+:` needs one `Term`, and then another, in order to produce an `Int`

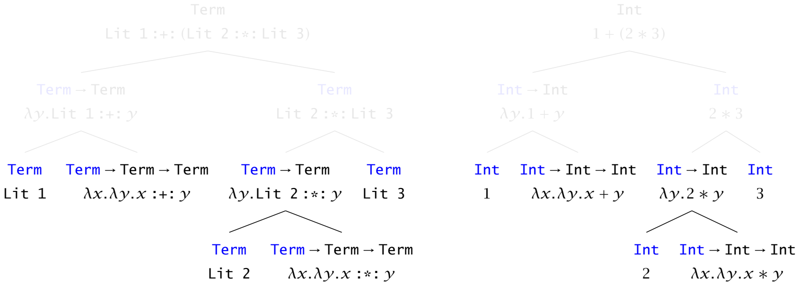
- So + is a **function** — a recipe for turning inputs to outputs — and it takes its inputs one at a time, making it **higher-order**.
- Functions represented with λ-calculus: if $f(x) = x^2$, we write f as $\lambda x.x^2$.

Term construction and evaluation



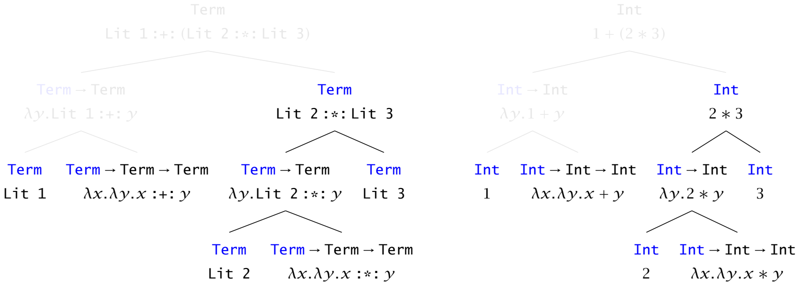
`Term` evaluation is compositional, homomorphic.

Term construction and evaluation



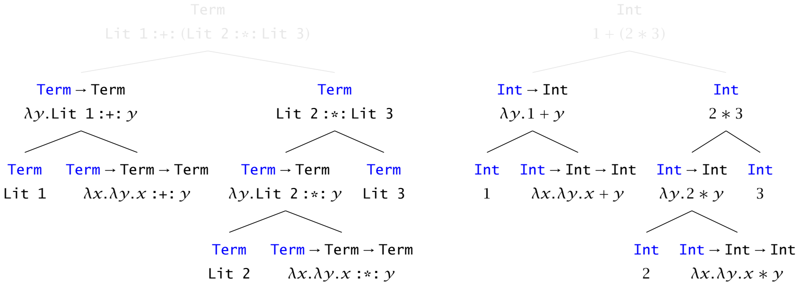
`Term` evaluation is compositional, homomorphic.

Term construction and evaluation



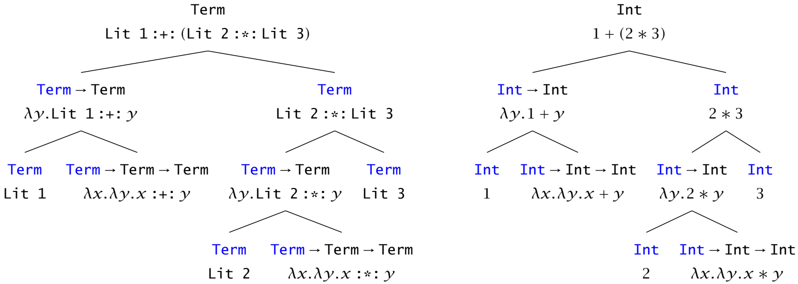
`Term` evaluation is compositional, homomorphic.

Term construction and evaluation



`Term` evaluation is compositional, homomorphic.

Term construction and evaluation



`Term` evaluation is compositional, homomorphic.

A baseline (extensional) semantic theory

Start with some basic types, and then ascend:[1]

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

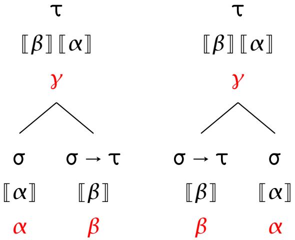
Interpret binary combination via application

Function Application

If a node γ has two daughters

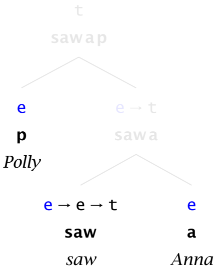
1. α of type σ, and
2. β of type $\sigma \to \tau$,

then γ has type τ, and $[\![\gamma]\!] = [\![\beta]\!]\,[\![\alpha]\!]$

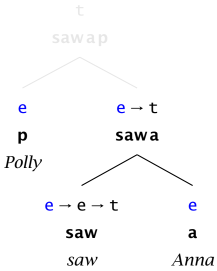


[1] $\mathtt{e}$ and $\mathtt{t}$ are the ι and o of Church's original Simple Theory of Types.

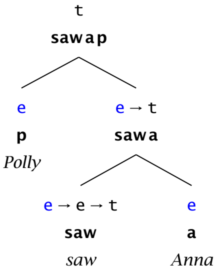
A couple examples



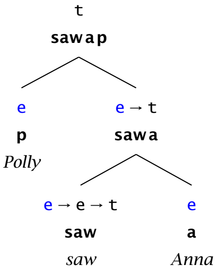
A couple examples

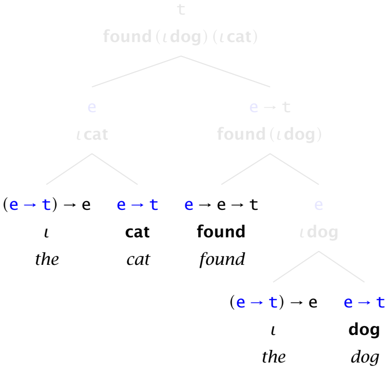


A couple examples

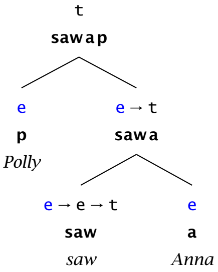


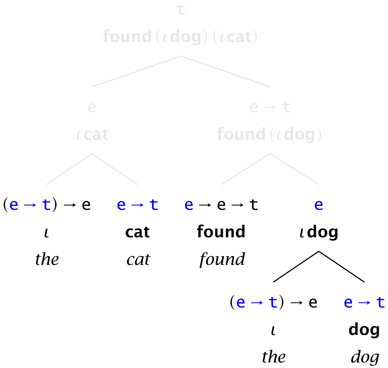
A couple examples



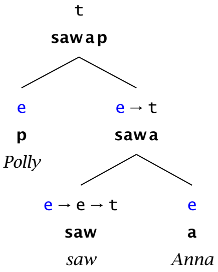


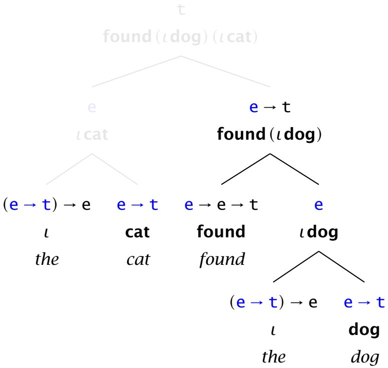
A couple examples



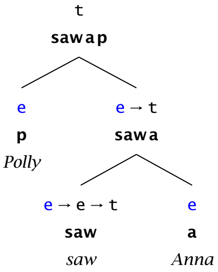


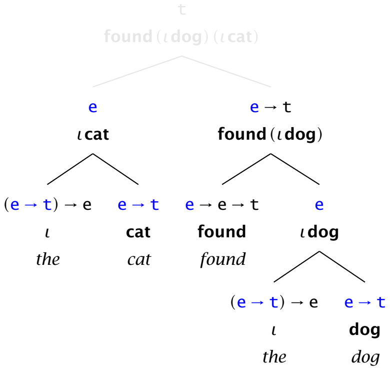
A couple examples



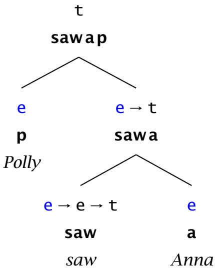


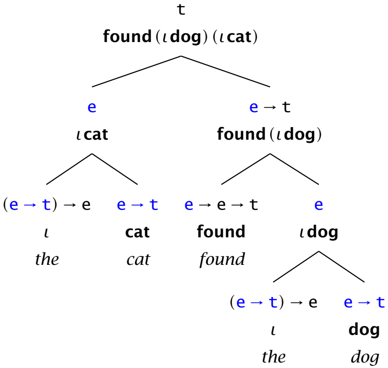
A couple examples





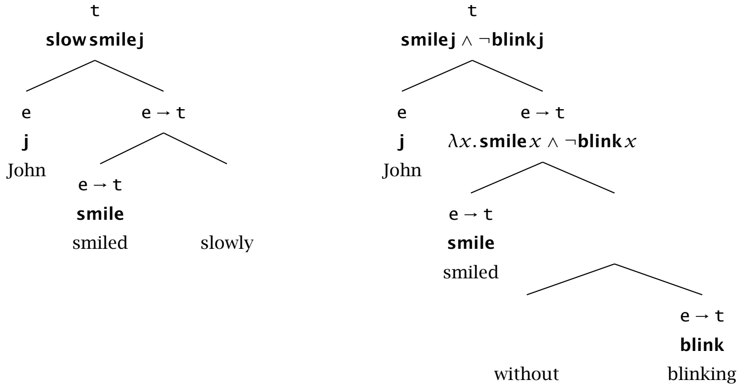
A couple examples





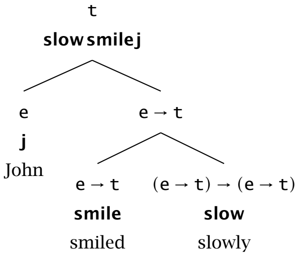
Running with Function Application

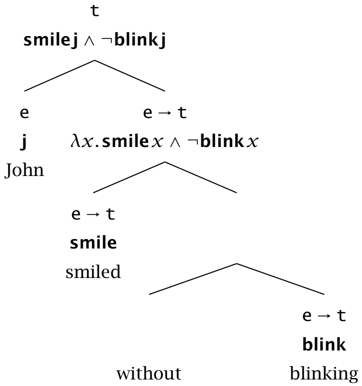
New words and constructions can always be assigned new denotations that fit into this picture



Running with Function Application

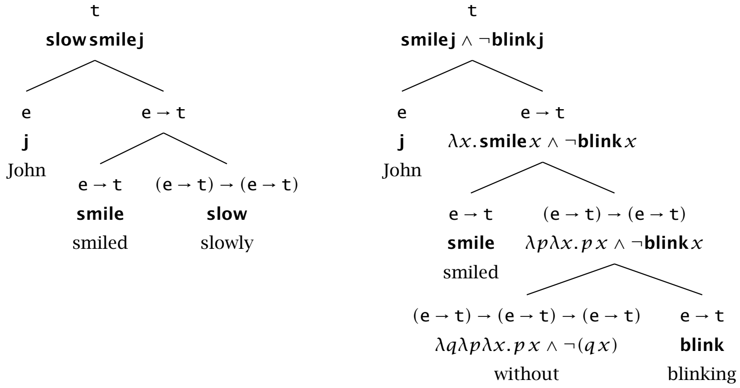
New words and constructions can always be assigned new denotations that fit into this picture





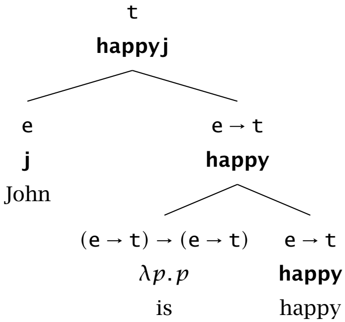
Running with Function Application

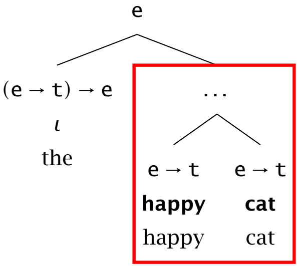
New words and constructions can always be assigned new denotations that fit into this picture



Stress test

Occasionally though, new configurations with **already-analyzed** language can lead to **type clashes**





New modes of combination

If the discrepancy is systematic enough, it can lead to proposals for additional **modes of combination**, e.g.,

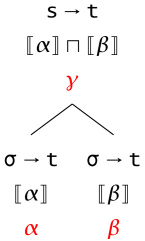
Predicate Modification

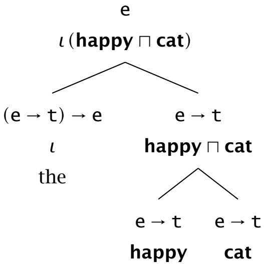
If a node γ has two daughters

1. α of type $\sigma \to t$, and
2. β of type $\sigma \to t$,

then γ has type $\sigma \to t$, and

$[\![\gamma]\!] = [\![\beta]\!] \sqcap [\![\alpha]\!]$





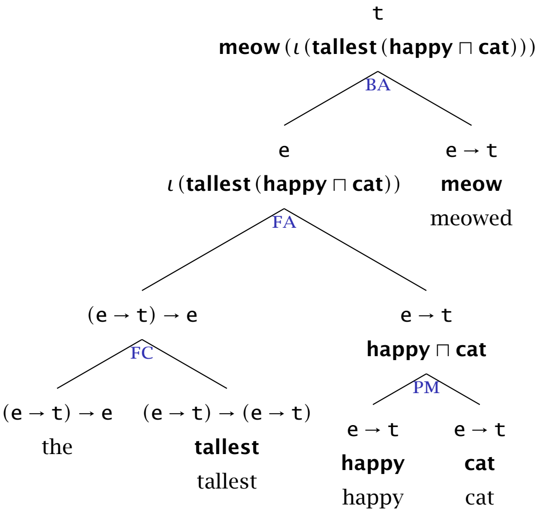
Type-driven composition

This brings us to more or less the standard picture (Klein & Sag 1985, Heim & Kratzer 1998):

- denotations built from a few basic kinds of objects, and functions over them
- a few basic modes of combination, with composition determined by types

$$[\![A\ B]\!] ::= \begin{cases} [\![A]\!]\,[\![B]\!] & \text{if } A :: \sigma \to \tau,\ B :: \sigma & \text{FA} \\ [\![B]\!]\,[\![A]\!] & \text{if } A :: \sigma,\ B :: \sigma \to \tau & \text{BA} \\ [\![A]\!] \cap [\![B]\!] & \text{if } A, B :: \sigma \to t & \text{PM} \\ [\![A]\!] \circ [\![B]\!] & \text{if } A :: \tau \to \upsilon,\ B :: \sigma \to \tau & \text{FC} \\ [\![B]\!] \upharpoonright [\![A]\!] & \text{if } A :: \sigma \to t,\ B :: \sigma \to \tau \to t & \text{PR} \\ \ldots & \text{if } \ldots & \ldots \end{cases}$$

D(e)riving with types

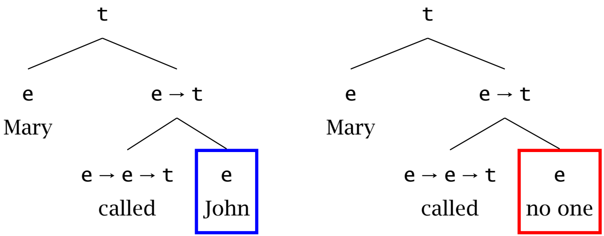


Effects

More than just an $\mathtt{e}$

This framework is extremely flexible, but some expressions seem to have too much meaning to fit into the sensible types

The most famous example of this comes from **quantificational noun phrases**

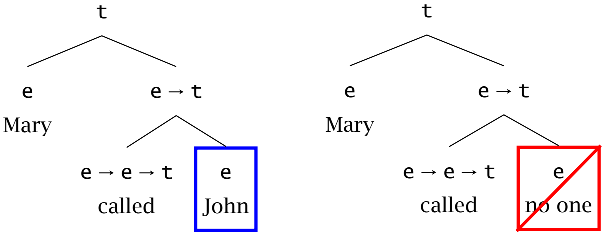


All reason suggests that ‘no one’ should have type $\mathtt{e}$ — it goes everywhere that ‘John’ goes — but there is no x s.t. $[\![\text{no one}]\!] = x$

More than just an e

This framework is extremely flexible, but some expressions seem to have too much meaning to fit into the sensible types

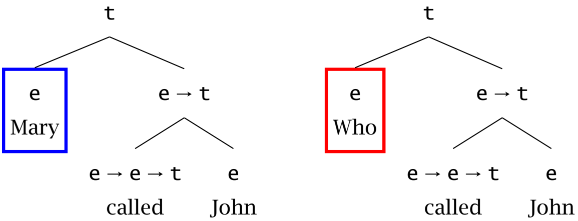
The most famous example of this comes from **quantificational noun phrases**



All reason suggests that ‘no one’ should have type e — it goes everywhere that ‘John’ goes — but there is no x s.t. $[\![\text{no one}]\!] = x$

More than just an e

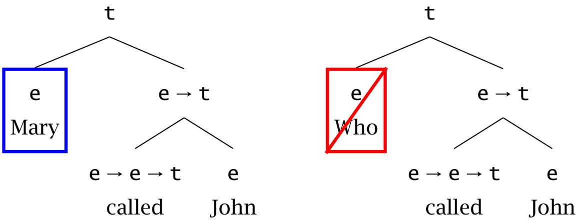
The same same might be said of **interrogative noun phrases**, like ‘who’



These clearly saturate the same argument positions as ordinary names, but clearly don’t refer to any particular entity

More than just an e

The same same might be said of **interrogative noun phrases**, like ‘who’



These clearly saturate the same argument positions as ordinary names, but clearly don’t refer to any particular entity

More than just an e

Less obviously, **indefinite noun phrases** like ‘a student’ seem to play the same compositional role as ordinary NPs

But like ‘wh’-words and quantifiers, they too clearly don’t name particular entities

Yet, unlike those extraordinary NPs, we do refer back to them as if they were names

1. Mary called. She was upset.
2. Someone called. She was upset.
3. # Everyone called. She was upset.

For that matter, **pronouns** are also very much like entity-like, without having stable referents. **Indexicals** too.

4. John saw her/me.

More than just an e

Definite descriptions also seem for all intents and purposes to denote entities ...

5. The two people teaching this class are American.

... except when they don't.

6. The three people teaching this class are American.

More than just an e

With a bit of **prosodic focus**, any noun phrase can be made to contribute more to what is said than its mere referent.

7. I only talked to John's sister.
8. I only talked to JOHN's sister.

Or you can always supplement the noun phrase with an **apposition**.

9. I talked to Mary, a first-year student.

Yet neither the focus nor the appositive change what kind of argument position the NP satisfies

Side Effects

All of these expressions have an outsized semantics relative to their compositional role, which is just that of an ordinay entity, e.

In this class we will take the view that these semantic enrichments should be treated as **(side) effects** of their evaluation

The inspiration here is from programming language theory

- 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)

Pure vs impure

For instance, consider a simple sort of program that changes the value of a variable while performing a computation

```
i = 0
print(i)
while i < 10:
  i += 1
print(i)
```

Here the meaning of the variable `i` depends on where in the program it is evaluated; it is in this sense **impure**

Haskell, like the lambda calculus and your typical natural language semantics, is **pure**: denotations are fixed, total functions from inputs to outputs.

How then can we think about the meanings of expressions that access and manipulate values in memory?

Effects and types

In both natural language semantics and functional programming, a guiding principle is that denotations should be **referentially transparent**

One facet of this is that if an expression's denotation isn't (merely) an entity, then its type can't (merely) be `e`

So a natural place to start is to decide what kinds of objects, and what kinds of types, these special NPs have

Algebraic Data Types

Some of these effects seem to call for denotations with multiple dimensions of meaning

- Sassy, a cat :: $\mathtt{e} \times \mathtt{t}$
  $[\![\text{Sassy, a cat}]\!] = \langle \mathbf{s}, \mathbf{cat}\,\mathbf{s} \rangle$

Other effects seem to call for denotations with multiple variants of meaning

- the cat :: $\mathtt{e} \mid \#$
  $[\![\text{the cat}]\!] = x$ if $\mathbf{cat} = \{x\}$ else #

Types built from these products, sums, and functions are called **Algebraic Data Types**

Some natural choices

Here are some other natural choices for effect types

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

Some natural choices

Here are some other natural choices for effect types

Expression	Type	Denotation
no cat	`(e -> Bool) -> Bool`	$\lambda c.\neg\exists x.\mathbf{cat}\,x \wedge c\,x$
which cat	`[e]`	$\{x \mid \mathbf{cat}\,x\}$
a cat	`s -> [(e, s)]`	$\lambda s.\{\langle x, s +\!\!+ x\rangle\}$
the cat	`e|#`	x if $\mathbf{cat} = \{x\}$ else #
SASSY	`(e, [e])`	$\langle \mathbf{s}, \{x \mid x \in D_{\mathsf{e}}\}\rangle$
Sassy, a cat	`(e, Bool)`	$\langle \mathbf{s}, \mathbf{cat}\,\mathbf{s}\rangle$
she	`r -> e`	$\lambda g.g_0$

Functors

Notice that in all of these, we have an `e` situated in some kind of **structural context**

Expression	Type
no cat	`( -> Bool) -> Bool`
which cat	`[ ]`
a cat	`s -> [( , s)]`
the cat	`|#`
SASSY	`( , [ ])`
Sassy, a cat	`( , Bool)`
she	`r ->`

These structural contexts are known in the Category- and Programming-Theory literatures as **Functors**

Functors

Notice that in all of these, we have an `e` situated in some kind of **structural context**

Expression	Type
no cat	`(e -> Bool) -> Bool`
which cat	`[e]`
a cat	`s -> [(e, s)]`
the cat	`e` |#
SASSY	`(e, [e])`
Sassy, a cat	`(e, Bool)`
she	`r -> e`

These structural contexts are known in the Category- and Programming-Theory literatures as **Functors**

Functor examples

Formally, we might think of a Functor as a function from types to types, e.g.

$$\mathtt{F}(\alpha) = \alpha \times \mathtt{t}$$

As it happens, many of the particular Functors in our table already have idiosyncratic names (or at least close approximations) in Haskell

Mathematical Type	Haskell Type
$(\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$	`data Cont t a = Cont ((a -> t) -> t)`
$\{\mathtt{e}\}$	`data [] a = [a]`
$\mathtt{s} \to \{\mathtt{e} \times \mathtt{s}\}$	`data State s a = State (s -> (a,s))`
$\mathtt{e} \mid \#$	`data Maybe a = Just a | Nothing`
$\mathtt{e} \times \mathtt{t}$	`data Writer t a = Writer a t`
$\mathtt{r} \to \mathtt{e}$	`data Reader r a = Reader (r -> a)`
$\mathtt{e} \times \{\mathtt{e}\}$	`--`

Functorial operations

However, not every function from types to types is a functor

The value(s) of type α hiding in the structure $\mathsf{F}(\alpha)$ must be, intuitively speaking, accessible to other operations

For instance, say you have a set of numbers, and a function to update those numbers

$$S = \{1, 2, 3\} \qquad f = \lambda n.\, n + 1$$

We can modify the numbers in S by **mapping** f over the contents of S

$$S' = \{f\,n \mid n \in S\} = \{f\,1,\ f\,2,\ f\,3\} = \{2, 3, 4\}$$

We would do the same thing if we started with a set of strings and wanted to update them by adding some text

$$S = \{\text{"a"}, \text{"b"}, \text{"c"}\} \qquad f = \lambda m.\, m \mathbin{+\!\!+} \text{"d"}$$

$$S' = \{f\,m \mid m \in S\} = \{f\,\text{"a"},\ f\,\text{"b"}, f\,\text{"c"}\} = \{\text{"ad"}, \text{"bd"}, \text{"cd"}\}$$

Functorial operations

Similarly if we have a number paired with a message, we can still easily modify the number by projecting it out and then pairing it back up

$$P = \langle 7, \text{"hello"} \rangle \qquad f = \lambda n.\, n + 1$$

$$P' = \langle f\, P_0, P_1 \rangle = \langle f\, 7, \text{"hello"} \rangle = \langle 8, \text{"hello"} \rangle$$

And again, we would do the same thing no matter what kind of data was stored in P

$$P = \langle \texttt{true}, \text{"hello"} \rangle \qquad f = \lambda b.\, \neg b$$

$$P' = \langle f\, P_0, P_1 \rangle = \langle f\, \texttt{true}, \text{"hello"} \rangle = \langle \texttt{false}, \text{"hello"} \rangle$$

No functorial operations

It might seem like this is trivial, but not every **structural context** guarantees accessibility like this

For instance, let an $\mathsf{N}(\alpha)$ be the type of a function that converts αs to numbers $\mathbb{N}$

Mathematical Type	Haskell Type
$\mathsf{N}(\alpha) = \alpha \to \mathbb{N}$	`data Encode a = Encode (a -> Int)`

Intuitively, there's no obvious sense in which something of type $\mathsf{N}(\alpha)$ is "storing" any αs in an accessible way

And indeed, imagine you have your hands on some function $E :: \mathsf{N}(\mathsf{t})$, together with a function f that can modify truth values; there's no way to use f to update E

$$E = \lambda b.\ \text{if } b \text{ then } 1 \text{ else } 0 \qquad f = \lambda b.\, \neg b$$

$$E' = \ldots f \ldots E \ldots ???$$

Functor laws

Technically, a type constructor F is a **Functor** if there is some operation

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

that will map a function $k :: \mathsf{a} \to \mathsf{b}$ over a structure $\mathsf{F}(\mathsf{a})$, yielding an $\mathsf{F}(\mathsf{b})$

In Haskell, this operation is called `fmap`

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

Moreover, the function should be reasonably well-behaved, satisfying the following two principles:

Identity:	$\mathbf{id} \bullet M = M$
Composition	$(f \circ g) \bullet M = (f \bullet (g \bullet M))$

Functor instances

For most Functors, these instances pretty much write themselves

$$\mathsf{W}(\alpha) ::= \alpha \times \mathsf{t}$$

$$k \bullet \langle a, b\rangle = \langle k\,a, b\rangle$$

```haskell
data Writer p a = Writer a p
instance Functor (Writer p) where
  fmap k (Writer a p) = Writer (k a) p
```

$$\mathsf{S}(\alpha) ::= \{\alpha\}$$

$$k \bullet S = \{k\,a \mid a \in S\}$$

```haskell
data [] a = [a]
instance Functor [] where
  fmap k as = [k a | a <- as]
```

In fact, it is literally impossible to write an instance of `fmap` that does not satisfy the **Composition** law (Wadler 1989)

Functor instances

```
data Maybe a = Just a | Nothing
instance Functor Maybe where
  fmap k m = case m of Nothing -> Nothing
                       Just a  -> Just (k a)
```

Functor instances

```haskell
data Maybe a = Just a | Nothing
instance Functor Maybe where
  fmap k m = case m of Nothing -> Nothing
                       Just a  -> Just (k a)
```

You can, however, if you try, write an `fmap` that does not satisfy **Identity**

```haskell
data Maybe a = Just a | Nothing
instance Functor Maybe where
  fmap k m = case m of Nothing -> Nothing
                       Just a  -> Nothing
```

```haskell
fmap id (Just 3)
  == case (Just 3) of Nothing -> Nothing
                      Just a  -> Nothing
  == Nothing
  /= (Just 3)
```

Functor instances

```haskell
data Reader r a = Reader (r -> a)

instance Functor (Reader r) where
  fmap k (Reader m) = Reader ...
```

Functor instances

```
data Reader r a = Reader (r -> a)

instance Functor (Reader r) where
  fmap k (Reader m) = Reader (\r -> k (m r))
```

Functor instances

```
data Reader r a = Reader (r -> a)

instance Functor (Reader r) where
  fmap k (Reader m) = Reader (\r -> k (m r))
```

With a little effort ...

```
data Cont t a = Cont ((a -> t) -> t)

instance Functor (Cont t) where
  fmap k (Cont m) = Cont ...
```

Functor instances

```haskell
data Reader r a = Reader (r -> a)

instance Functor (Reader r) where
  fmap k (Reader m) = Reader (\r -> k (m r))
```

With a little effort . . .

```haskell
data Cont t a = Cont ((a -> t) -> t)

instance Functor (Cont t) where
  fmap k (Cont m) = Cont (\c -> m (\a -> c (k a)))
```

Denotations in Functors

With these Functors in hand, we might rewrite our denotational table

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	$\mathtt{D\,e} ::= \mathtt{s} \to \{\mathtt{e} \times \mathtt{s}\}$	$\lambda s.\, \{\langle x, s \mathbin{+\!\!+} x \rangle \mid \mathbf{cat}\, x\}$
...	...	...

The type constructors bring out the sense in which all of these expressions essentially contribute type-$\mathtt{e}$ meanings, but also trigger particular effects

Composition again

Recall the problem we started with was that these effectful bits of language need to slot in where no effect is expected

$$
\begin{array}{cc}
\texttt{S}\,\texttt{e} & \texttt{e} \to \texttt{t} \\
\text{which cat} & \text{meowed}
\end{array}
$$

How does knowing that `S` is a Functor help? Well, we can now apply `fmap` to turn the VP into a function expecting an `S e` instead of an ordinary `e`

$$
\begin{array}{c}
\texttt{S}\,\texttt{t} \\
\textbf{meow} \bullet \{x \mid \textbf{cat}\, x\}
\end{array}
$$

$$
\begin{array}{cc}
\texttt{S}\,\texttt{e} & \texttt{S}\,\texttt{e} \to \texttt{S}\,\texttt{t} \\
\{x \mid \textbf{cat}\, x\} & \lambda S.\textbf{meow} \bullet S \\
\text{which cat} & \bullet \\
 & \texttt{e} \to \texttt{t} \\
 & \textbf{meow} \\
 & \text{meowed}
\end{array}
$$

Composition some more

And of course the same strategy will work for any Functorial effect

- $\texttt{M t}$: $\textbf{meow} \bullet (x \text{ if } \textbf{cat} = \{x\} \text{ else } \#)$
  - $\texttt{M e}$: $x \text{ if } \textbf{cat} = \{x\} \text{ else } \#$ — the cat
  - $\texttt{M e} \to \texttt{M t}$: $\lambda E.\, \textbf{meow} \bullet E$
    - $\bullet$
      - $\texttt{e} \to \texttt{t}$: $\textbf{meow}$ — meowed

- $\texttt{R t}$: $\textbf{meow} \bullet \lambda g.\, g_0$
  - $\texttt{R e}$: $\lambda g.\, g_0$ — she
  - $\texttt{R e} \to \texttt{R t}$: $\lambda E.\, \textbf{meow} \bullet E$
    - $\bullet$
      - $\texttt{e} \to \texttt{t}$: $\textbf{meow}$ — meowed

- $\texttt{W t}$: $\textbf{meow} \bullet \langle \textbf{s}, \textbf{cat s} \rangle$
  - $\texttt{W e}$: $\langle \textbf{s}, \textbf{cat s} \rangle$ — Sassy, a cat
  - $\texttt{W e} \to \texttt{W t}$: $\lambda E.\, \textbf{meow} \bullet E$
    - $\bullet$
      - $\texttt{e} \to \texttt{t}$: $\textbf{meow}$ — meowed

Composition some more

And of course the same strategy will work for any Functorial effect

$$\texttt{M}\,\texttt{t}$$
$$\textbf{meow} \bullet (x \text{ if } \textbf{cat} = \{x\} \text{ else } \#)$$
$$= (\textbf{meow}\, x) \text{ if } \textbf{cat} = \{x\} \text{ else } \#$$

$$\begin{array}{cc}
\texttt{M}\,\texttt{e} & \texttt{M}\,\texttt{e} \to \texttt{M}\,\texttt{t} \\
x \text{ if } \textbf{cat} = \{x\} \text{ else } \# & \lambda E.\,\textbf{meow} \bullet E \\
\text{the cat} & \Big|\bullet \\
 & \texttt{e} \to \texttt{t} \\
 & \textbf{meow} \\
 & \text{meowed}
\end{array}$$

$$\texttt{R}\,\texttt{t}$$
$$\textbf{meow} \bullet \lambda g.\, g_0$$

$$\begin{array}{cc}
\texttt{R}\,\texttt{e} & \texttt{R}\,\texttt{e} \to \texttt{R}\,\texttt{t} \\
\lambda g.\, g_0 & \lambda E.\,\textbf{meow} \bullet E \\
\text{she} & \Big|\bullet \\
 & \texttt{e} \to \texttt{t} \\
 & \textbf{meow} \\
 & \text{meowed}
\end{array}$$

$$\texttt{W}\,\texttt{t}$$
$$\textbf{meow} \bullet \langle \textbf{s}, \textbf{cat}\,\textbf{s} \rangle$$

$$\begin{array}{cc}
\texttt{W}\,\texttt{e} & \texttt{W}\,\texttt{e} \to \texttt{W}\,\texttt{t} \\
\langle \textbf{s}, \textbf{cat}\,\textbf{s} \rangle & \lambda E.\,\textbf{meow} \bullet E \\
\text{Sassy, a cat} & \Big|\bullet \\
 & \texttt{e} \to \texttt{t} \\
 & \textbf{meow} \\
 & \text{meowed}
\end{array}$$

Composition some more

And of course the same strategy will work for any Functorial effect

- $\texttt{M t}$
  $\mathbf{meow} \bullet (x \text{ if } \mathbf{cat} = \{x\} \text{ else } \#)$
  $= (\mathbf{meow}\, x) \text{ if } \mathbf{cat} = \{x\} \text{ else } \#$
  - $\texttt{M e}$
    $x \text{ if } \mathbf{cat} = \{x\} \text{ else } \#$
    the cat
  - $\texttt{M e} \to \texttt{M t}$
    $\lambda E.\, \mathbf{meow} \bullet E$
    - $\bullet$
      $\texttt{e} \to \texttt{t}$
      $\mathbf{meow}$
      meowed

- $\texttt{R t}$
  $\mathbf{meow} \bullet \lambda g.\, g_0$
  $= \lambda g.\, \mathbf{meow}\, g_0$
  - $\texttt{R e}$
    $\lambda g.\, g_0$
    she
  - $\texttt{R e} \to \texttt{R t}$
    $\lambda E.\, \mathbf{meow} \bullet E$
    - $\bullet$
      $\texttt{e} \to \texttt{t}$
      $\mathbf{meow}$
      meowed

- $\texttt{W t}$
  $\mathbf{meow} \bullet \langle \mathbf{s}, \mathbf{cat}\, \mathbf{s} \rangle$
  - $\texttt{W e}$
    $\langle \mathbf{s}, \mathbf{cat}\, \mathbf{s} \rangle$
    Sassy, a cat
  - $\texttt{W e} \to \texttt{W t}$
    $\lambda E.\, \mathbf{meow} \bullet E$
    - $\bullet$
      $\texttt{e} \to \texttt{t}$
      $\mathbf{meow}$
      meowed

Composition some more

And of course the same strategy will work for any Functorial effect

$\mathtt{M\,t}$
$\textbf{meow} \bullet (x \text{ if } \textbf{cat} = \{x\} \text{ else } \#)$
$= (\textbf{meow}\, x) \text{ if } \textbf{cat} = \{x\} \text{ else } \#$

- $\mathtt{M\,e}$
  $x \text{ if } \textbf{cat} = \{x\} \text{ else } \#$
  the cat
- $\mathtt{M\,e} \to \mathtt{M\,t}$
  $\lambda E.\, \textbf{meow} \bullet E$
  $\bullet$
  - $\mathtt{e} \to \mathtt{t}$
    meow
    meowed

$\mathtt{R\,t}$
$\textbf{meow} \bullet \lambda g.\, g_0$
$= \lambda g.\, \textbf{meow}\, g_0$

- $\mathtt{R\,e}$
  $\lambda g.\, g_0$
  she
- $\mathtt{R\,e} \to \mathtt{R\,t}$
  $\lambda E.\, \textbf{meow} \bullet E$
  $\bullet$
  - $\mathtt{e} \to \mathtt{t}$
    meow
    meowed

$\mathtt{W\,t}$
$\textbf{meow} \bullet \langle \textbf{s}, \textbf{cat}\, \textbf{s} \rangle$
$= \langle \textbf{meow}\, \textbf{s}, \textbf{cat}\, \textbf{s} \rangle$

- $\mathtt{W\,e}$
  $\langle \textbf{s}, \textbf{cat}\, \textbf{s} \rangle$
  Sassy, a cat
- $\mathtt{W\,e} \to \mathtt{W\,t}$
  $\lambda E.\, \textbf{meow} \bullet E$
  $\bullet$
  - $\mathtt{e} \to \mathtt{t}$
    meow
    meowed

Composition in more positions

Note that it is no more difficult to compose a verb with an effectful object than it has been with an effectful subject

$$\mathtt{W}(\mathtt{e} \to \mathtt{t})$$
$$\mathbf{see} \bullet \langle \mathbf{s}, \mathbf{cats} \rangle$$
$$= \langle \mathbf{see\,s}, \mathbf{cats} \rangle$$

$$\mathtt{We} \to \mathtt{W}(\mathtt{e} \to \mathtt{t}) \qquad \mathtt{We}$$
$$\lambda E.\,\mathbf{see} \bullet E \qquad \langle \mathbf{s}, \mathbf{cats} \rangle$$

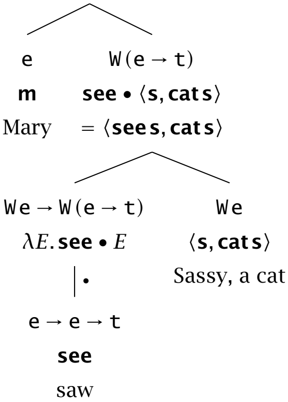
Sassy, a cat

$$\mathtt{e} \to \mathtt{e} \to \mathtt{t}$$
$$\mathbf{see}$$

saw

Composition in more problematic positions

However, there is a problem combining an effectful VP with an ordinary subject



Remember that $(\bullet)$ combines an ordinary function $k :: \mathtt{a} \to \mathtt{b}$ with an effectful argument $E :: \mathtt{F\,a}$, but we have the opposite

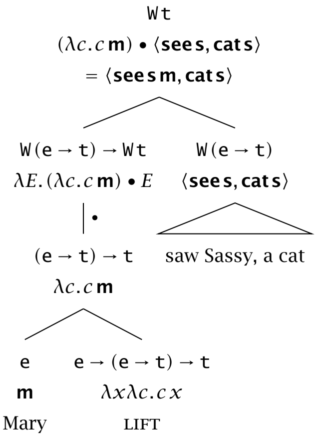
Lifting

To proceed, we might simply lean on the oldest trick in the semanticist's book: invert the function-argument relationship

Expression	Type	Denotation
LIFT	$\mathtt{a} \to (\mathtt{a} \to \mathtt{b}) \to \mathtt{b}$	$\lambda x \lambda c.\, c\, x$

With this, the ordinary argument becomes the ordinary function, and the effectful function becomes the effectful argument

So we `fmap` once more



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

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

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

no cat

$$\bullet$$

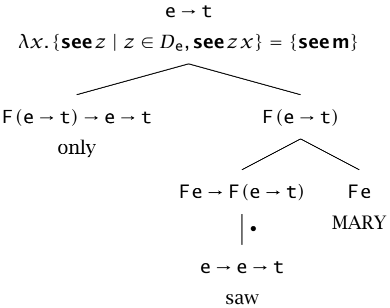
$$\texttt{e} \to \texttt{t}$$
$$\mathbf{meow}$$

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 $\blacksquare_{\mathsf{H}} :: \mathtt{H}\,\mathtt{t} \to \mathtt{t}$.

- A sentence with an environmental dependency is true if it is true in the utterance context (cf. Kaplan 1979)
  $\blacksquare_{\mathsf{R}} = \lambda v.\, v\, g_c$
- A sentence with a supplement is true only if both of its dimensions are true (cf. Boër & Lycan 1976)
  $\blacksquare_{\mathsf{W}} = \lambda \langle p, q \rangle.\, p \wedge q$
- 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)
  $\blacksquare_{\mathsf{M}} = \lambda m.\, \mathtt{false}$ 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)
  $\blacksquare_{\mathsf{S}} = \lambda S.\, \bigvee S$

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): $\blacksquare_{\mathsf{C}} = \lambda T.\, T\,\mathbf{id}$

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

$$\mathtt{C\,t \to t} \qquad\qquad \mathtt{C\,t}$$
$$\lambda T.\, T\,\mathbf{id} \qquad \mathbf{meow} \bullet (\lambda c.\,\neg\exists x.\,\mathbf{cat}\,x \wedge c\,x)$$
$$\blacksquare_{\mathsf{C}} \qquad = \lambda c.\,\neg\exists x.\,\mathbf{cat}\,x \wedge c\,(\mathbf{meow}\,x)$$

$$\mathtt{C\,e} \qquad\qquad \mathtt{C\,e \to C\,t}$$
$$\lambda c.\,\neg\exists x.\,\mathbf{cat}\,x \wedge c\,x \qquad \lambda E.\,\mathbf{meow} \bullet E$$

no cat

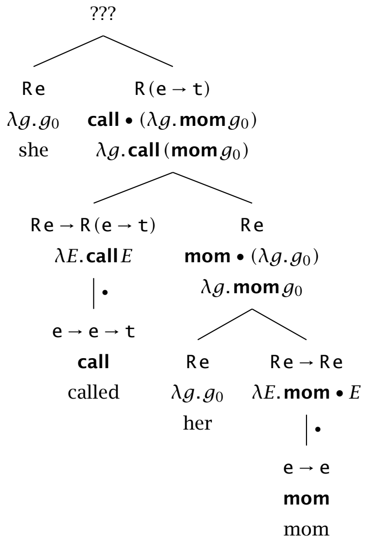
$$\bullet$$

$$\mathtt{e \to t}$$
$$\mathbf{meow}$$

meowed

Effects upon effects

We end with a challenge and a teaser for tomorrow: how to proceed with multiple, independent effectful components in the same derivation?



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

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.

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

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

Klein, Ewan & Ivan A. Sag. 1985. Type-driven translation. *Linguistics and Philosophy* 8(2). 163–201. https://doi.org/10.1007/BF00632365.

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.

Wadler, Philip. 1989. Theorems for free! In *Fourth international conference on functional programming languages and computer architecture*, 347–359.