Algebraic effects in Montague semantics

Julian Grove

CLASP, University of Gothenburg

October 28, 2020

Outline

1. Side effects in linguistic semantics
2. Algebraic effects and handlers
3. Making it Montagovian
4. Quantification and dynamism

We are here

Semantics is for…

Semantics

Semantics is for…

- characterizing semantic knowledge…

Semantics

Semantics is for...

- characterizing semantic knowledge...
  - ...i.e., knowledge of *entailments*? *distributional properties*?

Semantics is for…

- characterizing semantic knowledge…
  - …i.e., knowledge of *entailments*? *distributional properties*?
- describing how linguistic structure (i.e., syntax) gives rise to the things being characterized (whatever they are)

Semantics is for…

- characterizing semantic knowledge…
  - …i.e., knowledge of *entailments*? *distributional properties*?
- describing how linguistic structure (i.e., syntax) gives rise to the things being characterized (whatever they are)
- describing how pragmatic stuff (e.g., presupposing something, referring to something, expressing something) should affect the things being characterized

Montague

Montague

- used models as a vehicle to characterize meanings in terms of entailments

- used models as a vehicle to characterize meanings in terms of entailments
- described how linguistic structure gives rise to meanings, *compositionally*

Montague

- used models as a vehicle to characterize meanings in terms of entailments
- described how linguistic structure gives rise to meanings, *compositionally*
  - simply typed λ-calculus

- used models as a vehicle to characterize meanings in terms of entailments
- described how linguistic structure gives rise to meanings, *compositionally*
  - simply typed λ-calculus
- no pragmatic stuff

Montague 1973:

Rules of functional application

S4. If $\alpha \in \mathrm{P}_{\mathrm{t/IV}}$ and $\delta \in \mathrm{P}_{\mathrm{IV}}$, then $F_4(\alpha, \delta) \in \mathrm{P}_{\mathrm{t}}$, where $F_4(\alpha, \delta) = \alpha\delta'$ and δ' is the result of replacing the first *verb* (i.e., member of B_{IV}, B_{TV}, $\mathrm{B}_{\mathrm{IV/t}}$, or $\mathrm{B}_{\mathrm{IV//IV}}$) in δ by its third person singular present.

Rules of functional application

T4. If $\delta \in \mathrm{P}_{\mathrm{t/IV}}, \beta \in \mathrm{P}_{\mathrm{IV}}$, and δ, β translate into δ', β' respectively, then $F_4(\delta, \beta)$ translates into $\delta'(\hat{}\beta')$.

T5. If $\delta \in \mathrm{P}_{\mathrm{IV/T}}$, $\beta \in \mathrm{P}_{\mathrm{T}}$, and δ, β translate into δ', β' respectively, then $F_5(\delta, \beta)$

$$\begin{array}{ccc} & \lambda s.\mathbf{sleepj}s & \\ \text{John} \quad \text{slept} & \rightsquigarrow & \lambda P, s.Ps(\lambda s.\mathbf{j}) \quad \lambda s, x.\mathbf{sleep}(xs)s \end{array}$$

Quantifying in

Rules of quantification

S14. If $\alpha \in \mathrm{P}_\mathrm{T}$ and $\phi \in \mathrm{P}_\mathrm{t}$, then $F_{10,n}(\alpha, \phi) \in \mathrm{P}_\mathrm{t}$, where either (i) α does not have the form $\mathbf{he}_k$, and $F_{10,n}(\alpha, \phi)$ comes from ϕ by replacing the first occurrence of $\mathbf{he}_n$ or $\mathbf{him}_n$ by α and all other occurrences of $\mathbf{he}_n$ or $\mathbf{him}_n$ by $\left\{\begin{array}{l}\mathbf{he}\\ \mathbf{she}\\ \mathbf{it}\end{array}\right\}$ or $\left\{\begin{array}{l}\mathbf{him}\\ \mathbf{her}\\ \mathbf{it}\end{array}\right\}$ respectively, according as the gender of the first B_CN or B_T in α is $\left\{\begin{array}{l}\text{masc.}\\ \text{fem.}\\ \text{neuter}\end{array}\right\}$, or
(ii) $\alpha = \mathbf{he}_k$, and $F_{10,n}(\alpha, \phi)$ comes from ϕ by replacing all occurrences of $\mathbf{he}_n$ or $\mathbf{him}_n$ by $\mathbf{he}_k$ or $\mathbf{him}_k$ respectively.

Rules of quantification

T14. If $\alpha \in \mathrm{P}_\mathrm{T}$, $\phi \in \mathrm{P}_\mathrm{t}$, and α, ϕ translate into α', ϕ' respectively, then $F_{10,n}(\alpha, \phi)$ translates into $\alpha'(\hat{x}_n \phi')$.

Quantifying in

Every dog slept

$$\begin{array}{cc} \text{every dog} & \text{he}_n \text{ slept} \end{array} \quad \rightsquigarrow \quad \begin{array}{c} \lambda s.\forall u : \mathbf{dog}\, us \rightarrow \mathbf{sleep}\, \underline{u} s \\ \lambda P, s.\forall u : \mathbf{dog}\, us \rightarrow P(\lambda s.u)s \qquad \lambda s.\mathbf{sleep}(\underline{x_n} s)s \end{array}$$

Quantifying in

Every dog slept

every dog   he$_n$ slept $\rightsquigarrow$

$$\lambda s.\forall u : \mathbf{dog}\, u s \rightarrow \mathbf{sleep}\, \underline{u} s$$

$$\lambda P, s.\forall u : \mathbf{dog}\, u s \rightarrow P(\lambda s.u) s \qquad \lambda s.\mathbf{sleep}(\underline{x_n} s) s$$

Not compositional

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)
- Quantifier Raising and Predicate Abstraction (Heim and Kratzer, 1998)

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)
- Quantifier Raising and Predicate Abstraction (Heim and Kratzer, 1998)
- Flexible types and/or combinators (Hendriks, 1993; Steedman, 2000)

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)
- Quantifier Raising and Predicate Abstraction (Heim and Kratzer, 1998)
- Flexible types and/or combinators (Hendriks, 1993; Steedman, 2000)
- Grammars with interfaces to side effects

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)
- Quantifier Raising and Predicate Abstraction (Heim and Kratzer, 1998)
- Flexible types and/or combinators (Hendriks, 1993; Steedman, 2000)
- Grammars with interfaces to side effects
    - Continuations (Barker, 2002; Barker and Shan, 2014)

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)
- Quantifier Raising and Predicate Abstraction (Heim and Kratzer, 1998)
- Flexible types and/or combinators (Hendriks, 1993; Steedman, 2000)
- Grammars with interfaces to side effects
  - Continuations (Barker, 2002; Barker and Shan, 2014)
  - Monads (Shan, 2002; Charlow, 2014)

Since then

Many techniques since Montague for establishing seemingly non-local quantifier-variable dependencies…

- Cooper Storage and variants thereof (Cooper, 1983; Keller, 1988)
- Quantifier Raising and Predicate Abstraction (Heim and Kratzer, 1998)
- Flexible types and/or combinators (Hendriks, 1993; Steedman, 2000)
- Grammars with interfaces to side effects
  - Continuations (Barker, 2002; Barker and Shan, 2014)
  - Monads (Shan, 2002; Charlow, 2014)
  - Idioms (Kobele, 2018)

Side effects

Programming languages may exhibit "impure" behaviors.

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")
- partiality/exceptions

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")
- partiality/exceptions
- non-determinism

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")
- partiality/exceptions
- non-determinism

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")
- partiality/exceptions
- non-determinism

Pure programs merely manipulate data…

Side effects

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")
- partiality/exceptions
- non-determinism

Pure programs merely manipulate data...

- e.g., through functional application
  - ```
    def add(x, y):
        return (x + y)
    ```
    ```

Programming languages may exhibit "impure" behaviors.

- input/output
  - print "hi"
- environment/state
  - os.path.exists("./gremlins2.mov")
- partiality/exceptions
- non-determinism

Pure programs merely manipulate data...

- e.g., through functional application
  - def add(x, y):
        return (x + y)

Theories of side effects (e.g., monads) provide interfaces to impure behavior.

The effectful approach:

The effectful approach:

- identity linguistic phenomenon that appears to behave “impurely”, i.e., by subverting compositionality

The effectful approach:

- identity linguistic phenomenon that appears to behave "impurely", i.e., by subverting compositionality
  - e.g., quantification, anaphora, conventional implicature...

The effectful approach:

- identity linguistic phenomenon that appears to behave "impurely", i.e., by subverting compositionality
    - e.g., quantification, anaphora, conventional implicature…
- find an effectful interface that appropriately describes its behavior

Linguistic side effects

The effectful approach:

- identity linguistic phenomenon that appears to behave "impurely", i.e., by subverting compositionality
  - e.g., quantification, anaphora, conventional implicature...
- find an effectful interface that appropriately describes its behavior
- add it to your compositional repertoire!

This talk

- present two monadic interfaces to side effects: one for quantification and one anaphora

This talk

- present two monadic interfaces to side effects: one for quantification and one anaphora
  - the Continuation monad and the State monad, respectively

This talk

- present two monadic interfaces to side effects: one for quantification and one anaphora
  - the Continuation monad and the State monad, respectively
  - analyses inspired by Charlow (2014)

This talk

- present two monadic interfaces to side effects: one for quantification and one anaphora
  - the Continuation monad and the State monad, respectively
  - analyses inspired by Charlow (2014)
- show how they may and *may not* be combined

This talk

- present two monadic interfaces to side effects: one for quantification and one anaphora
  - the Continuation monad and the State monad, respectively
  - analyses inspired by Charlow (2014)
- show how they may and *may not* be combined
- introduce *algebraic effects*

Monads

a functor $\mathcal{M}$, equipped with two operators, $(\cdot)^\eta$ ('return') and $\gg\!=$ ('bind')

Definition ($\mathcal{M}$)

$$\mathcal{M} : \mathcal{T} \to \mathcal{T}$$
$$(\cdot)^\eta : a \to \mathcal{M}(b)$$
$$(\gg\!=) : \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

Monads

a functor $\mathcal{M}$, equipped with two operators, $(\cdot)^\eta$ ('return') and $\gg\!=$ ('bind')

Definition ($\mathcal{M}$)

$$\mathcal{M} : \mathcal{T} \to \mathcal{T}$$
$$(\cdot)^\eta : a \to \mathcal{M}(b)$$
$$(\gg\!=) : \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

Intuitively: $\mathcal{M}(a)$ is the space where the side effects of some value of type a happen.

a functor $\mathcal{M}$, equipped with two operators, $(\cdot)^\eta$ ('return') and $\gg=$ ('bind')

Definition ($\mathcal{M}$)

$$\mathcal{M} : \mathcal{T} \to \mathcal{T}$$
$$(\cdot)^\eta : a \to \mathcal{M}(b)$$
$$(\gg=) : \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

Intuitively: $\mathcal{M}(a)$ is the space where the side effects of some value of type a happen.

- $(\cdot)^\eta$ lifts pure values into that space.

a functor $\mathcal{M}$, equipped with two operators, $(\cdot)^\eta$ ('return') and $\gg\!=$ ('bind')

Definition ($\mathcal{M}$)

$$\mathcal{M} : \mathcal{T} \rightarrow \mathcal{T}$$
$$(\cdot)^\eta : a \rightarrow \mathcal{M}(b)$$
$$(\gg\!=) : \mathcal{M}(a) \rightarrow (a \rightarrow \mathcal{M}(b)) \rightarrow \mathcal{M}(b)$$

Intuitively: $\mathcal{M}(a)$ is the space where the side effects of some value of type a happen.

- $(\cdot)^\eta$ lifts pure values into that space.
- $\gg\!=$ sequences programs inhabiting that space by binding the result of one to the input of the next.

Monads

a functor $\mathcal{M}$, equipped with two operators, $(\cdot)^\eta$ ('return') and $\gg\!=$ ('bind')

Definition ($\mathcal{M}$)

$$\mathcal{M} : \mathcal{T} \to \mathcal{T}$$
$$(\cdot)^\eta : a \to \mathcal{M}(b)$$
$$(\gg\!=) : \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

Intuitively: $\mathcal{M}(a)$ is the space where the side effects of some value of type a happen.

- $(\cdot)^\eta$ lifts pure values into that space.
- $\gg\!=$ sequences programs inhabiting that space by binding the result of one to the input of the next.

Monads

a functor $\mathcal{M}$, equipped with two operators, $(\cdot)^\eta$ ('return') and $\gg\!=$ ('bind')

Definition ($\mathcal{M}$)

$$\mathcal{M} : \mathcal{T} \to \mathcal{T}$$
$$(\cdot)^\eta : a \to \mathcal{M}(b)$$
$$(\gg\!=) : \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

Intuitively: $\mathcal{M}(a)$ is the space where the side effects of some value of type a happen.

- $(\cdot)^\eta$ lifts pure values into that space.
- $\gg\!=$ sequences programs inhabiting that space by binding the result of one to the input of the next.

The operators must satisfy the Monad Laws.

The Monad Laws

$$v^{\eta} \gg\!= k = kv \tag{Left Identity}$$

$$m \gg\!= \lambda v.v^{\eta} = m \tag{Right Identity}$$

$$(m \gg\!= n) \gg\!= o = m \gg\!= \lambda v.nv \gg\!= o \tag{Associativity}$$

First case: quantification

In the Continuation monad, scope-taking is a kind of side effect.

Definition ($\mathcal{C}$)

$$\mathcal{C}(a) : (a \to o) \to o$$

In the Continuation monad, scope-taking is a kind of side effect.

Definition ($\mathcal{C}$)

$$\mathcal{C}(a) : (a \to o) \to o$$

$$(\cdot)^{\eta} : a \to (a \to o) \to o$$

In the Continuation monad, scope-taking is a kind of side effect.

Definition ($\mathcal{C}$)

$$\mathcal{C}(a) : (a \to o) \to o$$

$$(\cdot)^{\eta} : a \to (a \to o) \to o$$
$$v^{\eta} = \lambda c.cv$$

In the Continuation monad, scope-taking is a kind of side effect.

Definition ($\mathcal{C}$)

$$\mathcal{C}(a) : (a \to o) \to o$$

$$(\cdot)^{\eta} : a \to (a \to o) \to o$$
$$v^{\eta} = \lambda c.cv$$

$$(\ggg\!=) : ((a \to o) \to o) \to (a \to (b \to o) \to o) \to (b \to o) \to o$$

First case: quantification

In the Continuation monad, scope-taking is a kind of side effect.

Definition ($\mathcal{C}$)

$$\mathcal{C}(a) : (a \to o) \to o$$

$$(\cdot)^\eta : a \to (a \to o) \to o$$
$$v^\eta = \lambda c.cv$$

$$(\ggcurly\!=) : ((a \to o) \to o) \to (a \to (b \to o) \to o) \to (b \to o) \to o$$
$$m \ggcurly\!= k = \lambda c.m(\lambda v.kvc)$$

How it works

1. Ashley hugged every dog.

1. Ashley hugged every dog.

$$\begin{aligned}
\texttt{ashley} &= \mathbf{a}^{\eta} : \mathcal{C}(e) \\
\texttt{hugged} &= \mathbf{hug}^{\eta} : \mathcal{C}(e \to t) \\
\texttt{every} &= \lambda P, c.\forall x : Px \to cx : (e \to t) \to (e \to t) \to t \\
\texttt{dog} &= \mathbf{dog} : e \to t
\end{aligned} \qquad \text{(Lexicon)}$$

How it works

1. Ashley hugged every dog.

$$\begin{aligned} \texttt{ashley} &= \mathbf{a}^{\eta} : \mathcal{C}(e) \\ \texttt{hugged} &= \mathbf{hug}^{\eta} : \mathcal{C}(e \to t) \\ \texttt{every} &= \lambda P, c.\forall x : Px \to cx : (e \to t) \to \mathcal{C}(e) \\ \texttt{dog} &= \mathbf{dog} : e \to t \end{aligned} \tag{Lexicon}$$

How it works

1. Ashley hugged every dog.

$$\begin{aligned} \texttt{ashley} &= \mathbf{a}^{\eta} : \mathcal{C}(e) \\ \texttt{hugged} &= \mathbf{hug}^{\eta} : \mathcal{C}(e \to t) \\ \texttt{every} &= \lambda P, c.\forall x : Px \to cx : (e \to t) \to \mathcal{C}(e) \\ \texttt{dog} &= \mathbf{dog} : e \to t \end{aligned} \qquad \text{(Lexicon)}$$

$$\begin{aligned} (\triangleright) &: \mathcal{C}(a \to b) \to \mathcal{C}(a) \to \mathcal{C}(b) \\ m \triangleright n &= m \ggg\!= \lambda f.n \ggg\!= \lambda x.(f\,x)^{\eta} \\ &= \lambda c.m(\lambda f.n(\lambda x.c(f\,x))) \end{aligned} \qquad \text{(Grammar)}$$

$$\begin{aligned} (\triangleleft) &: \mathcal{C}(a) \to \mathcal{C}(a \to b) \to \mathcal{C}(b) \\ m \triangleleft n &= m \ggg\!= \lambda x.n \ggg\!= \lambda f.(f\,x)^{\eta} \\ &= \lambda c.m(\lambda x.n(\lambda f.c(f\,x))) \end{aligned}$$

1. Ashley hugged every dog.

$$\mathtt{ashley} \triangleleft (\mathtt{hugged} \triangleright \mathtt{everydog})$$

How it works

1. Ashley hugged every dog.

$$\mathbf{a}^{\eta} \triangleleft (\mathbf{hug}^{\eta} \triangleright \mathtt{every}\,\mathbf{dog})$$

1. Ashley hugged every dog.

$$\mathbf{a}^{\eta} \lhd (\mathbf{hug}^{\eta} \rhd \mathtt{every}\,\mathbf{dog})$$

expand $\mathtt{every}\,\mathbf{dog}$...

How it works

1. Ashley hugged every dog.

$$\mathbf{a}^{\eta} \lhd (\mathbf{hug}^{\eta} \rhd \lambda c.\forall x : \mathbf{dog}\, x \rightarrow c x)$$

1. Ashley hugged every dog.

$$\mathbf{a}^{\eta} \triangleleft (\mathbf{hug}^{\eta} \triangleright \lambda c.\forall x : \mathbf{dog}\, x \rightarrow c x)$$

$$\text{expand} \triangleright \ldots$$

How it works

1. Ashley hugged every dog.

$$\mathbf{a}^{\eta} \lhd \lambda c.\forall x : \mathbf{dog}\,x \to c(\mathbf{hug}\,x)$$

How it works

1. Ashley hugged every dog.

$$\mathbf{a}^{\eta} \triangleleft \lambda c.\forall x : \mathbf{dog}\, x \to c(\mathbf{hug}\, x)$$

$$\text{expand} \triangleleft \ldots$$

How it works

1. Ashley hugged every dog.

$$\lambda c.\forall x : \mathbf{dog}\, x \rightarrow c(\mathbf{hug}\, x\, \mathbf{a})$$

1. Ashley hugged every dog.

$$\lambda c.\forall x : \mathbf{dog}\,x \rightarrow c(\mathbf{hug}\,x\,\mathbf{a})$$

to obtain a proposition, apply to the identity function…

How it works

1. Ashley hugged every dog.

$$\forall x : \mathbf{dog}\,x \to \mathbf{hug}\,x\,\mathbf{a}$$

Summary

Using continuations to manage scope-taking:

Summary

Using continuations to manage scope-taking:

- scopal expressions take scope over their continuations, which are reified as they compose

Summary

Using continuations to manage scope-taking:

- scopal expressions take scope over their continuations, which are reified as they compose
- values take scope trivially (applying Montague's "lift")

In the State monad, we may read from and write to an environment.

Definition ($\mathcal{S}$)

$$\mathcal{S}(a) : s \to (s, a)$$

In the State monad, we may read from and write to an environment.

Definition ($\mathcal{S}$)

$$\mathcal{S}(a) : s \to (s, a)$$

$$(\cdot)^{\eta} : a \to s \to (s, a)$$

In the State monad, we may read from and write to an environment.

Definition ($\mathcal{S}$)

$$\mathcal{S}(a) : s \rightarrow (s, a)$$

$$(\cdot)^{\eta} : a \rightarrow s \rightarrow (s, a)$$

$$v^{\eta} = \lambda s.\langle s, v\rangle$$

In the State monad, we may read from and write to an environment.

Definition ($\mathcal{S}$)

$$\mathcal{S}(a) : s \to (s, a)$$

$$(\cdot)^\eta : a \to s \to (s, a)$$
$$v^\eta = \lambda s.\langle s, v\rangle$$

$$\begin{aligned}(\ggg\!=) : (s \to (s, a))& \\ \to (a \to s \to (s, b))& \\ \to s \to (s, b)&\end{aligned}$$

In the State monad, we may read from and write to an environment.

Definition ($\mathcal{S}$)

$$\mathcal{S}(a) : s \to (s, a)$$

$$(\cdot)^{\eta} : a \to s \to (s, a)$$
$$v^{\eta} = \lambda s.\langle s, v\rangle$$

$$(\gg\!=) : (s \to (s, a)) \to (a \to s \to (s, b)) \to s \to (s, b)$$
$$m \gg\!= k = \lambda s.\mathtt{let}\ \langle s', v\rangle = ms\ \mathtt{in}\ kvs'$$

How it works

1. Ashley hugged herself.

1. Ashley hugged herself.

$$\begin{aligned} \texttt{ashley} &= \mathbf{a}^{\eta} : \mathcal{S}(e) \\ \texttt{hugged} &= \mathbf{hug}^{\eta} : \mathcal{S}(e \rightarrow t) \\ \texttt{herself} &= \lambda s.\langle s, \texttt{sel}\, s\rangle : s \rightarrow (s, e) \end{aligned} \tag{Lexicon}$$

How it works

1. Ashley hugged herself.

$$\begin{aligned} \texttt{ashley} &= \mathbf{a}^{\eta} : \mathcal{S}(e) \\ \texttt{hugged} &= \mathbf{hug}^{\eta} : \mathcal{S}(e \to t) \\ \texttt{herself} &= \lambda s.\langle s, \texttt{sel}\, s\rangle : \mathcal{S}(e) \end{aligned} \tag{Lexicon}$$

How it works

1. Ashley hugged herself.

$$(\triangleright) : \mathcal{S}(a \to b) \to \mathcal{S}(a) \to \mathcal{S}(b) \qquad \text{(Grammar)}$$

$$\begin{aligned} m \triangleright n &= m \ggg \lambda f.n \ggg \lambda x.(f\,x)^{\eta} \\ &= \lambda s.\mathtt{let}\ \langle f, s' \rangle = ms\ \mathtt{in}\ \mathtt{let}\ \langle x, s'' \rangle = ns'\ \mathtt{in}\ \langle f\,x, s'' \rangle \end{aligned}$$

$$(\triangleleft) : \mathcal{S}(a) \to \mathcal{S}(a \to b) \to \mathcal{S}(b)$$

$$\begin{aligned} m \triangleleft n &= m \ggg \lambda x.n \ggg \lambda f.(f\,x)^{\eta} \\ &= \lambda s.\mathtt{let}\ \langle x, s' \rangle = ms\ \mathtt{in}\ \mathtt{let}\ \langle f, s'' \rangle = ns'\ \mathtt{in}\ \langle f\,x, s'' \rangle \end{aligned}$$

1. Ashley hugged herself.

$$
\begin{aligned}
(\triangleright) &: \mathcal{S}(a \to b) \to \mathcal{S}(a) \to \mathcal{S}(b) && \text{(Grammar)}\\
m \triangleright n &= m \ggg \lambda f.n \ggg \lambda x.(f\,x)^{\eta}\\
&= \lambda s.\texttt{let}\ \langle f, s' \rangle = ms\ \texttt{in}\ \texttt{let}\ \langle x, s'' \rangle = ns'\ \texttt{in}\ \langle f\,x, s'' \rangle\\
(\triangleleft) &: \mathcal{S}(a) \to \mathcal{S}(a \to b) \to \mathcal{S}(b)\\
m \triangleleft n &= m \ggg \lambda x.n \ggg \lambda f.(f\,x)^{\eta}\\
&= \lambda s.\texttt{let}\ \langle x, s' \rangle = ms\ \texttt{in}\ \texttt{let}\ \langle f, s'' \rangle = ns'\ \texttt{in}\ \langle f\,x, s'' \rangle\\
(\cdot)^{\blacktriangleright} &: \mathcal{S}(e) \to \mathcal{S}(e)\\
m^{\blacktriangleright} &= m \ggg \lambda x, s.\langle x{::}s, x \rangle
\end{aligned}
$$

How it works

1. Ashley hugged herself.

$$\texttt{ashley}^{\blacktriangleright} \lhd (\texttt{hugged} \rhd \texttt{herself})$$

How it works

1. Ashley hugged herself.

$$\mathtt{ashley}^{\blacktriangleright} \triangleleft (\mathbf{hug}^{\eta} \triangleright \mathtt{herself})$$

1. Ashley hugged herself.

$$(\lambda s.\langle \mathbf{a}::s, \mathbf{a}\rangle) \triangleleft (\mathbf{hug}^{\eta} \triangleright \mathtt{herself})$$

How it works

1. Ashley hugged herself.

$$(\lambda s.\langle \mathbf{a}{::}s, \mathbf{a}\rangle) \triangleleft (\mathbf{hug}^{\eta} \triangleright \lambda s.\langle s, \mathtt{sel}\, s\rangle)$$

How it works

1. Ashley hugged herself.

$$(\lambda s.\langle \mathbf{a}{::}s, \mathbf{a}\rangle) \lhd (\mathbf{hug}^{\eta} \rhd \lambda s.\langle s, \mathtt{sel}\, s\rangle)$$

$$\text{expand} \rhd \ldots$$

How it works

1. Ashley hugged herself.

$$(\lambda s.\langle \mathbf{a}{::}s, \mathbf{a}\rangle) \lhd \lambda s.\langle s, \mathbf{hug}(\mathtt{sel}\,s)\rangle$$

How it works

1. Ashley hugged herself.

$$(\lambda s.\langle \mathbf{a}{::}s, \mathbf{a}\rangle) \triangleleft \lambda s.\langle s, \mathbf{hug}(\texttt{sel}_1 s)\rangle$$

expand $\triangleleft \ldots$

How it works

1. Ashley hugged herself.

$$\lambda s.\langle \mathbf{a}{::}s, \mathbf{hug}(\mathtt{sel}(\mathbf{a}{::}s))\mathbf{a}\rangle$$

Summary

Using State to manage anaphora:

Summary

Using State to manage anaphora:

- expressions that introduce discourse referents or engage in anaphora engage with the environment

Summary

Using State to manage anaphora:

- expressions that introduce discourse referents or engage in anaphora engage with the environment
- values are trivially stateful, by passing the environment on, untouched

Combining quantification and anaphora

How might one do this?

How might one do this?

Answer: one may use *monad transformers* (the strategy adopted by Shan (2002), and then, by Charlow (2014)).

$\mathcal{C}$ and $\mathcal{S}$ are associated with corresponding monad transformers, $\mathcal{C}_T$ and $\mathcal{S}_T$.

Definition ($\mathcal{M}_T$)

$$\mathcal{M}_T : (\mathcal{T} \to \mathcal{T}) \to \mathcal{T} \to \mathcal{T}$$
$$(\cdot)^\eta : a \to \mathcal{M}_T(\mathcal{M}_0)(b)$$
$$(\gg\!=) : \mathcal{M}_T(\mathcal{M}_0)(a) \to (a \to \mathcal{M}_T(\mathcal{M}_0)(b)) \to \mathcal{M}_T(\mathcal{M}_0)(b)$$

given one of $\mathcal{C}$ or $\mathcal{S}$ as the *underlying monad*, we may apply one of $\mathcal{S}_T$ or $\mathcal{C}_T$ to it…

Definition ($\mathcal{C}_T$)

$$\mathcal{C}_T(\mathcal{M}_0)(a) : (a \to \mathcal{M}_0(o)) \to \mathcal{M}_0(o)$$

$$(\cdot)^\eta : a \to (a \to \mathcal{M}_0(o)) \to \mathcal{M}_0(o)$$
$$v^\eta = \lambda c.cv$$

$$\begin{aligned}(\ggg\!=) : ((a \to \mathcal{M}_0(o)) \to \mathcal{M}_0(o)) \\ \to (a \to (b \to \mathcal{M}_0(o)) \to \mathcal{M}_0(o)) \\ \to (b \to \mathcal{M}_0(o)) \to \mathcal{M}_0(o)\end{aligned}$$
$$m \ggg\!= k = \lambda c.m(\lambda v.kvc)$$

The State monad transformer

Definition ($\mathcal{S}_T$)

$$\mathcal{S}_T(\mathcal{M}_0)(a) : s \to \mathcal{M}_0((s, a))$$

$$(\cdot)^\eta : a \to s \to \mathcal{M}_0((s, a))$$
$$v^\eta = \lambda s.\langle s, v\rangle^\eta$$

$$\begin{aligned}(\gg\!=) &: (s \to \mathcal{M}_0((s, a))) \\ &\to (a \to (s \to \mathcal{M}_0((s, b)))) \\ &\to s \to \mathcal{M}_0((s, b))\end{aligned}$$
$$m \gg\!= k = \lambda s.ms \gg\!= \lambda p.\mathtt{let}\ \langle s', v\rangle = p\ \mathtt{in}\ kvs'$$

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

It turns out we must apply $\mathcal{C}_T$ to $\mathcal{S}$ and not $\mathcal{S}_T$ to $\mathcal{C}$.

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

It turns out we must apply $\mathcal{C}_T$ to $\mathcal{S}$ and not $\mathcal{S}_T$ to $\mathcal{C}$.

If we adopt the transformers approach from the start…

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

It turns out we must apply $\mathcal{C}_T$ to $\mathcal{S}$ and not $\mathcal{S}_T$ to $\mathcal{C}$.

If we adopt the transformers approach from the start…

- we throw out our generalized quantifier meaning for *every dog*

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

It turns out we must apply $\mathcal{C}_T$ to $\mathcal{S}$ and not $\mathcal{S}_T$ to $\mathcal{C}$.

If we adopt the transformers approach from the start…

- we throw out our generalized quantifier meaning for *every dog*
- the type of *every dog* becomes $(e \rightarrow \mathcal{M}_0(t)) \rightarrow \mathcal{M}_0(t)$ …

To summarize…

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

It turns out we must apply $\mathcal{C}_T$ to $\mathcal{S}$ and not $\mathcal{S}_T$ to $\mathcal{C}$.

If we adopt the transformers approach from the start…

- we throw out our generalized quantifier meaning for *every dog*
- the type of *every dog* becomes $(e \to \mathcal{M}_0(t)) \to \mathcal{M}_0(t)$ …
  - but a generic meaning for *every* cannot be written…we are required to know what $\mathcal{M}_0$ is!

To summarize...

This general strategy can be made to work extremely well (Charlow, 2014).

But, how do we decide which monad transformer to apply to which monad?

1. Every dog licked Ashley. *It is friendly.

It turns out we must apply $\mathcal{C}_T$ to $\mathcal{S}$ and not $\mathcal{S}_T$ to $\mathcal{C}$.

If we adopt the transformers approach from the start...

- we throw out our generalized quantifier meaning for *every dog*
- the type of *every dog* becomes $(e \to \mathcal{M}_0(t)) \to \mathcal{M}_0(t)$ ...
  - but a generic meaning for *every* cannot be written... we are required to know what $\mathcal{M}_0$ is!
  - even then, the meaning the quantifier will be somewhat stipulative, e.g., to account for the data above (though, it can be made to follow from a small set of primitives, as in Charlow (2014))

The problem

The transformers approach, when used generically, prevents us from writing meanings. When used non-generically, it loses extensibility.

The problem

The transformers approach, when used generically, prevents us from writing meanings. When used non-generically, it loses extensibility.

Might we salvage our individual analyses in some other way? In doing so, might we account for data like (1)?

We are here

Background

Algebraic effects and handlers provide a means of writing extensible code, recently especially popular in functional programming.[1]

[1] Original insights about the relation between algebra and computational effects are from Plotkin and Power 2001, 2003.

Background

Algebraic effects and handlers provide a means of writing extensible code, recently especially popular in functional programming.[1]

Jirka Maršík has done significant work importing algebraic effects into linguistic semantics, culminating in his dissertation (Maršík and Amblard, 2014, 2016; Maršík, 2016)

[1]Original insights about the relation between algebra and computational effects are from Plotkin and Power 2001, 2003.

Background

Algebraic effects and handlers provide a means of writing extensible code, recently especially popular in functional programming.[1]

Jirka Maršík has done significant work importing algebraic effects into linguistic semantics, culminating in his dissertation (Maršík and Amblard, 2014, 2016; Maršík, 2016)

- develops a typed extension of λ-calculus

[1] Original insights about the relation between algebra and computational effects are from Plotkin and Power 2001, 2003.

Background

Algebraic effects and handlers provide a means of writing extensible code, recently especially popular in functional programming.[1]

Jirka Maršík has done significant work importing algebraic effects into linguistic semantics, culminating in his dissertation (Maršík and Amblard, 2014, 2016; Maršík, 2016)

- develops a typed extension of λ-calculus
- studies an array of phenomena algebraically, including quantification, presupposition, conventional implicature, and deixis

[1] Original insights about the relation between algebra and computational effects are from Plotkin and Power 2001, 2003.

Background

Algebraic effects and handlers provide a means of writing extensible code, recently especially popular in functional programming.[1]

Jirka Maršík has done significant work importing algebraic effects into linguistic semantics, culminating in his dissertation (Maršík and Amblard, 2014, 2016; Maršík, 2016)

- develops a typed extension of λ-calculus
- studies an array of phenomena algebraically, including quantification, presupposition, conventional implicature, and deixis
- anaphora is approached using the compositional DRT of de Groote (2006)

[1] Original insights about the relation between algebra and computational effects are from Plotkin and Power 2001, 2003.

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically
- and then combining the resulting algebras

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically
- and then combining the resulting algebras

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically
- and then combining the resulting algebras

I will take a different approach from Maršík, by…

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically
- and then combining the resulting algebras

I will take a different approach from Maršík, by…

- staying in STLC (with unit)

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically
- and then combining the resulting algebras

I will take a different approach from Maršík, by…

- staying in STLC (with unit)
- characterizing anaphora in purely algebraic terms

The basic idea

Instead of fixing a monad transformer stack, we may study the side effects of individual phenomena independently…

- by characterizing them algebraically
- and then combining the resulting algebras

I will take a different approach from Maršík, by…

- staying in STLC (with unit)
- characterizing anaphora in purely algebraic terms
- sticking with a traditional analysis of quantifiers, i.e., whereon they denote sets of sets

An algebraic signature is a set E of operations, each one associated with a *parameter* p and an *arity* a (both types), along with a special operation η ('return').

$$E = \{\mathrm{op}_{1_{p_1 \rightsquigarrow a_1}}, \ldots, \mathrm{op}_{n_{p_n \rightsquigarrow a_n}}, \eta\}$$

An algebraic signature is a set E of operations, each one associated with a *parameter* p and an *arity* a (both types), along with a special operation η ('return').

$$E = \{\mathrm{op}_{1_{p_1 \rightsquigarrow a_1}}, \ldots, \mathrm{op}_{n_{p_n \rightsquigarrow a_n}}, \eta\}$$

Elements of the algebra with signature E inhabit a type which we call $\mathcal{F}_E(v)$ (for some *return* type v).

An algebraic signature is a set E of operations, each one associated with a *parameter* p and an *arity* a (both types), along with a special operation η ('return').

$$E = \{\mathrm{op}_{1_{p_1 \rightsquigarrow a_1}}, \dots, \mathrm{op}_{n_{p_n \rightsquigarrow a_n}}, \eta\}$$

Elements of the algebra with signature E inhabit a type which we call $\mathcal{F}_E(v)$ (for some *return* type v).

To say operator $\mathrm{op}_{p \rightsquigarrow a}$ is in signature E means that it has the following type signature:

$$\mathrm{op}_{p \rightsquigarrow a} : p \to (a \to \mathcal{F}_E(v)) \to \mathcal{F}_E(v)$$

Algebraic signatures

An algebraic signature is a set E of operations, each one associated with a *parameter* p and an *arity* a (both types), along with a special operation η ('return').

$$E = \{\mathrm{op}_{1_{p_1 \rightsquigarrow a_1}}, \ldots, \mathrm{op}_{n_{p_n \rightsquigarrow a_n}}, \eta\}$$

Elements of the algebra with signature E inhabit a type which we call $\mathcal{F}_E(v)$ (for some *return* type v).

To say operator $\mathrm{op}_{p \rightsquigarrow a}$ is in signature E means that it has the following type signature:

$$\mathrm{op}_{p \rightsquigarrow a} : p \to (a \to \mathcal{F}_E(v)) \to \mathcal{F}_E(v)$$

η always has the following type signature:

$$\eta : v \to \mathcal{F}_E(v)$$

Algebraic laws

In addition to the signature, an algebra determines a set of equations that must hold among its elements, of the form

$$\mathrm{op}_i(p_i; \ldots) = \mathrm{op}_j(p_j; \ldots)$$

The State algebra (signature)

instead of a State monad, we will have a State algebra

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\texttt{get}_{\star \rightsquigarrow s}$ and $\texttt{put}_{s \rightsquigarrow \star}$

- s is the type of the state

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

- s is the type of the state
- $\star$ is the unit type (one inhabitant, also called $\star$)

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

- s is the type of the state
- $\star$ is the unit type (one inhabitant, also called $\star$)

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

- s is the type of the state
- $\star$ is the unit type (one inhabitant, also called $\star$)

Some example elements of the State algebra…

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

- s is the type of the state
- $\star$ is the unit type (one inhabitant, also called $\star$)

Some example elements of the State algebra…

- $\mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\eta s) : \mathcal{F}_{\{\mathtt{get}_{\star \rightsquigarrow s}, \mathtt{put}_{s \rightsquigarrow \star}\}}(s)$

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

- s is the type of the state
- $\star$ is the unit type (one inhabitant, also called $\star$)

Some example elements of the State algebra…

- $\mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\eta s) : \mathcal{F}_{\{\mathtt{get}_{\star \rightsquigarrow s}, \mathtt{put}_{s \rightsquigarrow \star}\}}(s)$
- $\mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s \rightsquigarrow \star}(\mathbf{a}{::}s; \lambda \star .\eta s)) : \mathcal{F}_{\{\mathtt{get}_{\star \rightsquigarrow s}, \mathtt{put}_{s \rightsquigarrow \star}\}}(s)$

The State algebra (signature)

instead of a State monad, we will have a State algebra

two operations, $\mathtt{get}_{\star\rightsquigarrow s}$ and $\mathtt{put}_{s\rightsquigarrow\star}$

- s is the type of the state
- $\star$ is the unit type (one inhabitant, also called $\star$)

Some example elements of the State algebra…

- $\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\eta s) : \mathcal{F}_{\{\mathtt{get}_{\star\rightsquigarrow s}, \mathtt{put}_{s\rightsquigarrow\star}\}}(s)$
- $\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s\rightsquigarrow\star}(\mathbf{a}{::}s; \lambda\star.\eta s)) : \mathcal{F}_{\{\mathtt{get}_{\star\rightsquigarrow s}, \mathtt{put}_{s\rightsquigarrow\star}\}}(s)$
- $\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s\rightsquigarrow\star}(\mathbf{a}{::}s; \lambda\star.\eta\mathbf{a})) : \mathcal{F}_{\{\mathtt{get}_{\star\rightsquigarrow s}, \mathtt{put}_{s\rightsquigarrow\star}\}}(e)$

The State algebra (laws)

Reading the environment twice is no better than reading it once:

$$\texttt{get}_{\star \rightsquigarrow s}(\star; \lambda g.\texttt{get}_{\star \rightsquigarrow s}(\star; \lambda g'.kgg')) = \texttt{get}_{\star \rightsquigarrow s}(\star; \lambda g.kgg)$$

The State algebra (laws)

Reading the environment twice is no better than reading it once:

$$\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g.\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g'.kgg')) = \mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g.kgg)$$

Putting something back where you got it is the same as doing nothing:

$$\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g.\mathtt{put}_{s\rightsquigarrow\star}(g; k))) = k\star$$

The State algebra (laws)

Reading the environment twice is no better than reading it once:

$$\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g.\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g'.kgg')) = \mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g.kgg)$$

Putting something back where you got it is the same as doing nothing:

$$\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda g.\mathtt{put}_{s\rightsquigarrow\star}(g; k))) = k\star$$

Getting after you just put means getting what you put:

$$\mathtt{put}_{s\rightsquigarrow\star}(g; \lambda\star.\mathtt{get}_{\star\rightsquigarrow s}(\star; k)) = \mathtt{put}_{s\rightsquigarrow\star}(g; \lambda\star.kg)$$

The State algebra (laws)

Reading the environment twice is no better than reading it once:

$$\texttt{get}_{\star\rightsquigarrow s}(\star; \lambda g.\texttt{get}_{\star\rightsquigarrow s}(\star; \lambda g'.kgg')) = \texttt{get}_{\star\rightsquigarrow s}(\star; \lambda g.kgg)$$

Putting something back where you got it is the same as doing nothing:

$$\texttt{get}_{\star\rightsquigarrow s}(\star; \lambda g.\texttt{put}_{s\rightsquigarrow\star}(g; k))) = k\star$$

Getting after you just put means getting what you put:

$$\texttt{put}_{s\rightsquigarrow\star}(g; \lambda\star.\texttt{get}_{\star\rightsquigarrow s}(\star; k)) = \texttt{put}_{s\rightsquigarrow\star}(g; \lambda\star.kg)$$

Putting twice overwrites:

$$\texttt{put}_{s\rightsquigarrow\star}(g; \lambda\star.\texttt{put}_{s\rightsquigarrow\star}(g'; k)) = \texttt{put}_{s\rightsquigarrow\star}(g'; k)$$

The Quantifier algebra (signature)

one operation, $\text{scope}_{(e \to t) \to t \rightsquigarrow e}$

The Quantifier algebra (signature)

one operation, $\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}$

Some example elements of the Quantifier algebra…

- $\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}(\mathtt{every}\,\mathbf{dog}; \lambda y.\eta(\mathbf{sleep}\,y)) : \mathcal{F}_{\{\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}\}}(t)$
- $\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}(\mathtt{every}\,\mathbf{dog}; \lambda y.\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}(\mathtt{every}\,\mathbf{cat}; \lambda z.\eta(\mathbf{chase}\,z\,y))) : \mathcal{F}_{\{\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}\}}(t)$

Quantifying in:

$$\mathrm{scope}_{(e \to t) \to t \rightsquigarrow e}(q; \lambda x.\eta(kx)) = \eta(qk)$$

Combing the algebras…

is just a matter of

Combing the algebras…

is just a matter of

- collecting the operations into one signature

Combing the algebras…

is just a matter of

- collecting the operations into one signature
- combining the equations

Combing the algebras…

is just a matter of

- collecting the operations into one signature
- combining the equations
- adding one more law to allow $\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}$ to commute with $\mathtt{get}_{\star\rightsquigarrow s}$ and $\mathtt{put}_{s\rightsquigarrow\star}$

Combing the algebras…

is just a matter of

- collecting the operations into one signature
- combining the equations
- adding one more law to allow $\mathtt{scope}_{(e \to t) \to t \rightsquigarrow e}$ to commute with $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

is just a matter of

- collecting the operations into one signature
- combining the equations
- adding one more law to allow $\mathtt{scope}_{(e \to t) \to t \rightsquigarrow e}$ to commute with $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$

Commuting $\mathtt{scope}_{(e \to t) \to e \rightsquigarrow e}$ past $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{s \rightsquigarrow \star}$:

$$\mathtt{scope}_{(e \to t) \to e \rightsquigarrow e}(q; \lambda x.\mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s \rightsquigarrow \star}(s'; \lambda \star .kxss')))$$

$$= \mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s \rightsquigarrow \star}(s; \lambda \star .\mathtt{scope}_{(e \to t) \to e \rightsquigarrow e}(q; \lambda x.kxss')))$$

We are here

How to do it

What we want is an encoding of the operations, as well as a way of *translating* λ-terms with lots of operations into ones with fewer operations in a way that respects the algebraic laws.

How to do it

What we want is an encoding of the operations, as well as a way of *translating* λ-terms with lots of operations into ones with fewer operations in a way that respects the algebraic laws.

This is called “handling” the operations. It can treat algebraic laws essentially as *reduction rules*. From this perspective, we may obtain a “normal form” for algebraic elements.

How to do it

What we want is an encoding of the operations, as well as a way of *translating* λ-terms with lots of operations into ones with fewer operations in a way that respects the algebraic laws.

This is called "handling" the operations. It can treat algebraic laws essentially as *reduction rules*. From this perspective, we may obtain a "normal form" for algebraic elements.

In the combined State/Quantifier algebra, the normal form for any element is determined by the laws to be

$$\mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{\star \rightsquigarrow s}(f\, s; \eta(g s)))$$

for some $f : s \to s$ and $g : s \to v$.

How to do it

What we want is an encoding of the operations, as well as a way of *translating* λ-terms with lots of operations into ones with fewer operations in a way that respects the algebraic laws.

This is called "handling" the operations. It can treat algebraic laws essentially as *reduction rules*. From this perspective, we may obtain a "normal form" for algebraic elements.

In the combined State/Quantifier algebra, the normal form for any element is determined by the laws to be

$$\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{\star\rightsquigarrow s}(f\,s; \eta(g s)))$$

for some $f : s \to s$ and $g : s \to v$.

Pairs of such functions f and g can be represented as $\lambda s.\langle f\,s, g s\rangle$ ...they are State monadic!

Encoding elements

To encode elements of an algebra, we define a family of functors $\mathcal{F} : \mathcal{T}^*_{\leadsto} \to \mathcal{T} \to \mathcal{T}$, where

- $\mathcal{T}^*_{\leadsto}$ is the free monoid (i.e., of lists) over $\mathcal{T}_{\leadsto} = \{p \leadsto a \mid p, a \in \mathcal{T}\}$

Encoding elements

To encode elements of an algebra, we define a family of functors $\mathcal{F} : \mathcal{T}^*_{\rightsquigarrow} \to \mathcal{T} \to \mathcal{T}$, where

- $\mathcal{T}^*_{\rightsquigarrow}$ is the free monoid (i.e., of lists) over $\mathcal{T}_{\rightsquigarrow} = \{p \rightsquigarrow a \mid p, a \in \mathcal{T}\}$

$$\mathcal{F}_{\epsilon}(v) = v$$

$$\mathcal{F}_{p \rightsquigarrow a, l}(v) = (p \to (a \to \mathcal{F}_l(v)) \to o) \to o$$

To encode elements of an algebra, we define a family of functors $\mathcal{F} : \mathcal{T}^*_{\rightsquigarrow} \to \mathcal{T} \to \mathcal{T}$, where

- $\mathcal{T}^*_{\rightsquigarrow}$ is the free monoid (i.e., of lists) over $\mathcal{T}_{\rightsquigarrow} = \{p \rightsquigarrow a \mid p, a \in \mathcal{T}\}$

$$
\begin{aligned}
\mathcal{F}_{\epsilon}(v) &= v \\
\mathcal{F}_{p \rightsquigarrow a, l}(v) &= (p \to (a \to \mathcal{F}_l(v)) \to o) \to o
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{op}_{p \rightsquigarrow a} &: p \to (a \to \mathcal{F}_l(v)) \to \mathcal{F}_{p \rightsquigarrow a, l} \\
\mathrm{op}_{p \rightsquigarrow a}(p; k) &= \lambda h.hpk
\end{aligned}
$$

Encoding elements

To encode elements of an algebra, we define a family of functors $\mathcal{F} : \mathcal{T}^*_{\rightsquigarrow} \to \mathcal{T} \to \mathcal{T}$, where

- $\mathcal{T}^*_{\rightsquigarrow}$ is the free monoid (i.e., of lists) over $\mathcal{T}_{\rightsquigarrow} = \{p \rightsquigarrow a \mid p, a \in \mathcal{T}\}$

$$\mathcal{F}_{\epsilon}(v) = v$$
$$\mathcal{F}_{p \rightsquigarrow a, l}(v) = (p \to (a \to \mathcal{F}_l(v)) \to o) \to o$$

$$\mathrm{op}_{p \rightsquigarrow a} : p \to (a \to \mathcal{F}_l(v)) \to \mathcal{F}_{p \rightsquigarrow a, l}$$
$$\mathrm{op}_{p \rightsquigarrow a}(p; k) = \lambda h.hpk$$

$$\eta : v \to \mathcal{F}_{\epsilon}(v)$$
$$\eta v = v$$

To encode elements of an algebra, we define a family of functors $\mathcal{F} : \mathcal{T}^*_{\rightsquigarrow} \to \mathcal{T} \to \mathcal{T}$, where

- $\mathcal{T}^*_{\rightsquigarrow}$ is the free monoid (i.e., of lists) over $\mathcal{T}_{\rightsquigarrow} = \{p \rightsquigarrow a \mid p, a \in \mathcal{T}\}$

$$\mathcal{F}_{\epsilon}(v) = v$$
$$\mathcal{F}_{p \rightsquigarrow a, l}(v) = (p \to (a \to \mathcal{F}_l(v)) \to o) \to o$$

$$\mathrm{op}_{p \rightsquigarrow a} : p \to (a \to \mathcal{F}_l(v)) \to \mathcal{F}_{p \rightsquigarrow a, l}$$
$$\mathrm{op}_{p \rightsquigarrow a}(p; k) = \lambda h.hpk$$

$$\eta : v \to \mathcal{F}_{\epsilon}(v)$$
$$\eta v = v$$

Operations construct “pairs”; returning just returns…

An element of the State/Quantifier algebra

1. Every dog hugged itself.

An element of the State/Quantifier algebra

1. Every dog hugged itself.

$$\begin{aligned}&\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}(\mathtt{every}\,\mathbf{dog};\\ &\qquad \lambda x.\mathtt{get}_{\star\rightsquigarrow s}(\star;\\ &\qquad\qquad \lambda s.\mathtt{put}_{s\rightsquigarrow\star}(x{::}s;\\ &\qquad\qquad\qquad \lambda\star.\mathtt{get}_{\star\rightsquigarrow s}(\star;\lambda s'.\eta(\mathbf{hug}(\mathtt{sel}\,s')x)))))\end{aligned}$$

1. Every dog hugged itself.

$$\begin{aligned}&\mathtt{scope}_{(e\to t)\to t\rightsquigarrow e}(\mathtt{every}\,\mathbf{dog};\\&\qquad \lambda x.\mathtt{get}_{\star\rightsquigarrow s}(\star;\\&\qquad\qquad \lambda s.\mathtt{put}_{s\rightsquigarrow\star}(x{::}s;\\&\qquad\qquad\qquad \lambda\star.\mathtt{get}_{\star\rightsquigarrow s}(\star;\lambda s'.\eta(\mathbf{hug}(\mathtt{sel}\,s')x)))))\end{aligned}$$

$$= \lambda h.h(\mathtt{every}\,\mathbf{dog})(\lambda x,h'.h'\star(\lambda s.\ldots))$$

1. Every dog hugged itself.

$$
\begin{aligned}
&\texttt{scope}_{(e\to t)\to t\rightsquigarrow e}(\texttt{every}\,\mathbf{dog};\\
&\qquad \lambda x.\texttt{get}_{\star\rightsquigarrow s}(\star;\\
&\qquad\qquad \lambda s.\texttt{put}_{s\rightsquigarrow\star}(x{::}s;\\
&\qquad\qquad\qquad \lambda\star.\texttt{get}_{\star\rightsquigarrow s}(\star;\lambda s'.\eta(\mathbf{hug}(\texttt{sel}\,s')x)))))
\end{aligned}
$$

$$
= \lambda h.h(\texttt{every}\,\mathbf{dog})(\lambda x, h'.h' \star (\lambda s.\ldots))
$$

This will be an expression of type

$$
\begin{aligned}
&\mathcal{F}_{(e\to t)\to t\rightsquigarrow e,\star\rightsquigarrow s,s\rightsquigarrow\star,\star\rightsquigarrow s}\\
= \;&(((e\to t)\to t)\to(e\to((\star\to(s\to\ldots)\to o')\to o'))\to o)\to o
\end{aligned}
$$

Handling operations

We have a way of encoding meanings involving quantifiers and anaphora.

Handling operations

We have a way of encoding meanings involving quantifiers and anaphora.

What we would like is to provide a *handler* that implements our reduction rules, i.e., those determined by the algebraic laws.

We have a way of encoding meanings involving quantifiers and anaphora.

What we would like is to provide a *handler* that implements our reduction rules, i.e., those determined by the algebraic laws.

We need a family of functions

$$\texttt{handleSentence}_l : \mathcal{F}_l(t) \to \mathcal{F}_{\star \to s, s, \to \star}(t)$$

where $l \in \{(e \to t) \to t \rightsquigarrow e,\ \ \star \rightsquigarrow s,\ \ s \rightsquigarrow \star\}^*$.

We are here

Predictions

We would like to explain contrasts such as

1. Every dog licked Ashley. *It is friendly.
2. Ashley hugged every dog. She is friendly.
3. Every dog licked itself.

Predictions

We would like to explain contrasts such as

1. Every dog licked Ashley. *It is friendly.
2. Ashley hugged every dog. She is friendly.
3. Every dog licked itself.

When applied to the meanings of the initial sentences, $\texttt{handleSentence}_l$ delivers:

Predictions

We would like to explain contrasts such as

1. Every dog licked Ashley. *It is friendly.
2. Ashley hugged every dog. She is friendly.
3. Every dog licked itself.

When applied to the meanings of the initial sentences, $\texttt{handleSentence}_l$ delivers:

- $\texttt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\texttt{put}_{s \rightsquigarrow \star}(s; \lambda \star .\eta(\forall x : \mathbf{dog}\,x \rightarrow \mathbf{lick}\,\mathbf{a}\,x)))$

We would like to explain contrasts such as

1. Every dog licked Ashley. *It is friendly.
2. Ashley hugged every dog. She is friendly.
3. Every dog licked itself.

When applied to the meanings of the initial sentences, $\mathtt{handleSentence}_l$ delivers:

- $\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s\rightsquigarrow\star}(s; \lambda\star.\eta(\forall x : \mathbf{dog}\,x \to \mathbf{lick}\,\mathbf{a}\,x)))$
- $\mathtt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s\rightsquigarrow\star}(\mathbf{a}{::}s; \lambda\star.\eta(\forall x : \mathbf{dog}\,x \to \mathbf{hug}\,x\,\mathbf{a})))$

Predictions

We would like to explain contrasts such as

1. Every dog licked Ashley. *It is friendly.
2. Ashley hugged every dog. She is friendly.
3. Every dog licked itself.

When applied to the meanings of the initial sentences, $\texttt{handleSentence}_l$ delivers:

- $\texttt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\texttt{put}_{s \rightsquigarrow \star}(s; \lambda \star.\eta(\forall x : \mathbf{dog}\, x \to \mathbf{lick}\, \mathbf{a}\, x)))$
- $\texttt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\texttt{put}_{s \rightsquigarrow \star}(\mathbf{a} :: s; \lambda \star.\eta(\forall x : \mathbf{dog}\, x \to \mathbf{hug}\, x\, \mathbf{a})))$
- $\texttt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\texttt{put}_{s \rightsquigarrow \star}(s; \lambda \star.\eta(\forall x : \mathbf{dog}\, x \to \mathbf{lick}(\texttt{sel}(x :: s))x)))$

In sum

Our algebraic laws predict the contrasts! Crucial is the law that commutes $\mathtt{scope}_{(e\to t)\to t \rightsquigarrow e}$ past $\mathtt{get}_{\star\rightsquigarrow s}$ and $\mathtt{put}_{\star\rightsquigarrow s}$.

In sum

Our algebraic laws predict the contrasts! Crucial is the law that commutes $\texttt{scope}_{(e\to t)\to t\rightsquigarrow e}$ past $\texttt{get}_{\star\rightsquigarrow s}$ and $\texttt{put}_{\star\rightsquigarrow s}$.

$$\texttt{scope}_{(e\to t)\to e\rightsquigarrow e}(q; \lambda x.\texttt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\texttt{put}_{s\rightsquigarrow\star}(s'; \lambda\star.kxss')))$$

$$= \texttt{get}_{\star\rightsquigarrow s}(\star; \lambda s.\texttt{put}_{s\rightsquigarrow\star}(s; \lambda\star.\texttt{scope}_{(e\to t)\to e\rightsquigarrow e}(q; \lambda x.kxss')))$$

In sum

Our algebraic laws predict the contrasts! Crucial is the law that commutes $\mathtt{scope}_{(e \to t) \to t \rightsquigarrow e}$ past $\mathtt{get}_{\star \rightsquigarrow s}$ and $\mathtt{put}_{\star \rightsquigarrow s}$.

$$\mathtt{scope}_{(e \to t) \to e \rightsquigarrow e}(q; \lambda x.\mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s \rightsquigarrow \star}(s'; \lambda \star .kxss')))$$

$$= \mathtt{get}_{\star \rightsquigarrow s}(\star; \lambda s.\mathtt{put}_{s \rightsquigarrow \star}(s; \lambda \star .\mathtt{scope}_{(e \to t) \to e \rightsquigarrow e}(q; \lambda x.kxss')))$$

This law destroys a quantifier's dynamic potential, rendering it externally static.

Conclusion

The algebraic effects approach allows us to write semantic analyses which

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)
- are extensible (unlike monad transformers, where providing meanings came at the cost of expanding the grammar)

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)
- are extensible (unlike monad transformers, where providing meanings came at the cost of expanding the grammar)
- are relatively conservative (e.g., quantifiers are still of type $(e \rightarrow t) \rightarrow t$)

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)
- are extensible (unlike monad transformers, where providing meanings came at the cost of expanding the grammar)
- are relatively conservative (e.g., quantifiers are still of type $(e \rightarrow t) \rightarrow t$)
- allow us to study interactions between linguistic side effects, in terms of algebraic laws

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)
- are extensible (unlike monad transformers, where providing meanings came at the cost of expanding the grammar)
- are relatively conservative (e.g., quantifiers are still of type $(e \to t) \to t$)
- allow us to study interactions between linguistic side effects, in terms of algebraic laws

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)
- are extensible (unlike monad transformers, where providing meanings came at the cost of expanding the grammar)
- are relatively conservative (e.g., quantifiers are still of type $(e \to t) \to t$)
- allow us to study interactions between linguistic side effects, in terms of algebraic laws

This gives us a new and precise way of characterizing certain old semantic problems about quantification and dynamism:

Conclusion

The algebraic effects approach allows us to write semantic analyses which

- are compositional, using traditional tools (like, e.g., monads do)
- are extensible (unlike monad transformers, where providing meanings came at the cost of expanding the grammar)
- are relatively conservative (e.g., quantifiers are still of type $(e \to t) \to t$)
- allow us to study interactions between linguistic side effects, in terms of algebraic laws

This gives us a new and precise way of characterizing certain old semantic problems about quantification and dynamism:

- when combining algebras, where do any new laws come from? can they come for free?

References

Barker, Chris. 2002. Continuations and the Nature of Quantification. *Natural Language Semantics* 10:211–242. https://doi.org/10.1023/A:1022183511876.

Barker, Chris, and Chung-chieh Shan. 2014. *Continuations and natural language*, volume 53. Oxford studies in theoretical linguistics.

Charlow, Simon. 2014. On the semantics of exceptional scope. PhD Thesis, NYU, New York. https://semanticsarchive.net/Archive/2JmMWRjY.

Cooper, R. 1983. *Quantification and Syntactic Theory*. Studies in Linguistics and Philosophy. Springer Netherlands. https://www.springer.com/gp/book/9789027714848.

References

de Groote, Philippe. 2006. Towards a Montagovian Account of Dynamics. *Semantics and Linguistic Theory* 16:1–16. https://journals.linguisticsociety.org/proceedings/index.php/SALT/article/view/2952, number: 0.

Heim, Irene, and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Malden: Blackwell.

Hendriks, H. L. W. 1993. *Studied flexibility : categories and types in syntax and semantics*. AmsterdamInstitute for Logic, Language and Computation. https://dare.uva.nl/search?identifier=2a784df2-19f2-4d8a-8096-6a4c05db0316.

References

Keller, William R. 1988. Nested Cooper Storage: The Proper Treatment of Quantification in Ordinary Noun Phrases. In *Natural Language Parsing and Linguistic Theories*, ed. U. Reyle and C. Rohrer, Studies in Linguistics and Philosophy, 432–447. Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-009-1337-0_15.

Kobele, Gregory M. 2018. The Cooper Storage Idiom. *Journal of Logic, Language and Information* 27:95–131. https://doi.org/10.1007/s10849-017-9263-1.

Maršík, Jirka, and Maxime Amblard. 2016. Introducing a Calculus of Effects and Handlers for Natural Language Semantics. In *Formal Grammar*, ed. Annie Foret, Glyn Morrill, Reinhard Muskens, Rainer Osswald, and Sylvain Pogodalla, Lecture Notes in Computer Science, 257–272. Berlin, Heidelberg: Springer.

References

Maršík, Jiří. 2016. Effects and handlers in natural language. phdthesis, Université de Lorraine. https://hal.inria.fr/tel-01417467.

Maršík, Jiří, and Maxime Amblard. 2014. Algebraic Effects and Handlers in Natural Language Interpretation. In *Natural Language and Computer Science*, ed. Valeria de Paiva, Walther Neuper, Pedro Quaresma, Christian Retoré, Lawrence S. Moss, and Jordi Saludes, volume TR 2014-002 of *Joint Proceedings of the Second Workshop on Natural Language and Computer Science (NLCS'14) & 1st International Workshop on Natural Language Services for Reasoners (NLSR 2014)*. Vienne, Austria: Center for Informatics and Systems of the University of Coimbra. https://hal.archives-ouvertes.fr/hal-01079206.

References

Montague, Richard. 1973. The Proper Treatment of Quantification in Ordinary English. In *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, ed. K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes, Synthese Library, 221–242. Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-010-2506-5_10.

Plotkin, Gordon, and John Power. 2001. Semantics for Algebraic Operations. *Electronic Notes in Theoretical Computer Science* 45:332–345. http://www.sciencedirect.com/science/article/pii/S1571066104809708.

Plotkin, Gordon, and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11:69–94. https://doi.org/10.1023/A:1023064908962.

Shan, Chung-chieh. 2002. Monads for natural language semantics. *arXiv:cs/0205026* `http://arxiv.org/abs/cs/0205026`, arXiv: cs/0205026.

Steedman, Mark. 2000. *The syntactic process*, volume 24. MIT press Cambridge, MA.