Algebraic effects in Montague semantics

Julian Grove

CLASP, University of Gothenburg

October 28, 2020

Outline

- Side effects in linguistic semantics
- Algebraic effects and handlers
- Making it Montagovian
- Quantification and anaphora

We are here

- Side effects in linguistic semantics
- 2 Algebraic effects and handlers
- Making it Montagovian
- Quantification and anaphora

Semantics is for...

• characterizing semantic knowledge...

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

- 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)

- 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





 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



- 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

Functional application

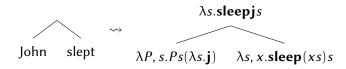
Montague 1973:

Rules of functional application

S4. If $\alpha \in P_{t/IV}$ and $\delta \in P_{IV}$, then $F_4(\alpha, \delta) \in P_t$, 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} , $B_{IV/t}$, or $B_{IV//IV}$) in δ by its third person singular present.

Rules of functional application

- T4. If $\delta \in P_{1/IV}$, $\beta \in P_{IV}$, and δ, β translate into δ', β' respectively, then $F_4(\delta, \beta)$ translates into $\delta'(\hat{\beta}')$.
- T5 If $\delta \in P_{m,m}$ $\beta \in P_m$ and δ β translate into δ' β' respectively then $F_{\epsilon}(\delta,\beta)$



Quantifying in

Rules of quantification

S14. If $\alpha \in P_T$ and $\phi \in P_t$, then $F_{10,n}(\alpha,\phi) \in P_t$, 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

$$\begin{array}{c} \text{him}_{n} \text{ by } \left\{ \begin{array}{c} \text{he} \\ \text{she} \\ \text{it} \end{array} \right\} \text{ or } \left\{ \begin{array}{c} \text{him} \\ \text{her} \\ \text{it} \end{array} \right\} \text{ respectively, according as the gender of the} \\ \text{first } B_{CN} \text{ or } B_{T} \text{ in } \alpha \text{ is } \left\{ \begin{array}{c} \text{masc.} \\ \text{fem.} \\ \text{powter} \end{array} \right\}, \text{ or} \end{array}$$

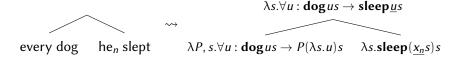
(ii) $\alpha = \mathbf{he}_k$, and $F_{10,n}(\alpha, \phi)$ comes from ϕ by replacing all occurrences of \mathbf{he}_n or \mathbf{him}_n by \mathbf{he}_k or \mathbf{him}_k respectively.

Rules of quantification

T14. If $\alpha \in P_T$, $\phi \in P_t$, and α , ϕ translate into α' , ϕ' respectively, then $F_{10,n}(\alpha,\phi)$ translates into $\alpha'(\hat{x}_n\phi')$.

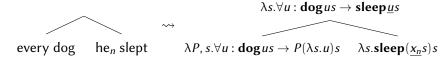
Quantifying in

Every dog slept



Quantifying in

Every dog slept



Not compositional

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

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

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

- 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)

- 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

- 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)

- 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)

- 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)

Programming languages may exhibit "impure" behaviors.

input/output

- input/output
 - print "hi"

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

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

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

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

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

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...

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:

• identity linguistic phenomenon that appears to behave "impurely", i.e., by subverting compositionality

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

- 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

- 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!

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

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

- present two monadic interfaces to side effects: one for quantification and one anaphora
 - the Continuation monad and the State monad, respectively
- show how they may and may not be combined

- present two monadic interfaces to side effects: one for quantification and one anaphora
 - the Continuation monad and the State monad, respectively
- show how they may and may not be combined
- introduce algebraic effects

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

Definition (M)

$$\mathcal{M}: \mathcal{T} \to \mathcal{T}$$

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

$$(\gg): \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

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: 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: $\mathfrak{M}(a)$ 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 (
$$\mathcal{M}$$
)
$$\mathcal{M}: \mathcal{T} \to \mathcal{T}$$

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

$$(\gg): \mathcal{M}(a) \to (a \to \mathcal{M}(b)) \to \mathcal{M}(b)$$

Intuitively: M(a) is the space where the side effects of some value of type a happen.

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

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: M(a) is the space where the side effects of some value of type a happen.

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

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

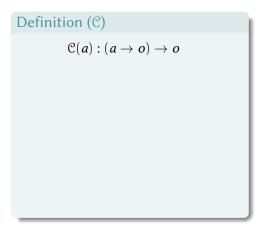
- $(\cdot)^{\eta}$ lifts pure values into that space.
- sequences programs inhabiting that space by binding the result of one to the input of the next.

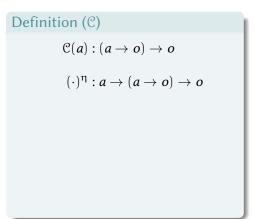
The operators must satisfy the Monad Laws.

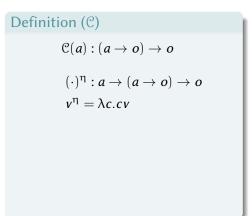


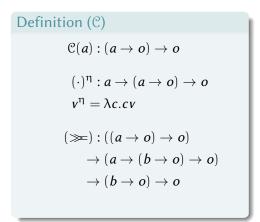
The Monad Laws

$$v^{\eta} \gg k = kv$$
 (Left Identity) $m \gg \lambda v. v^{\eta} = m$ (Right Identity) $(m \gg n) \gg o = m \gg \lambda v. nv \gg o$ (Associativity)









Definition (C)
$$C(a): (a \to o) \to o$$

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

$$v^{\eta} = \lambda c. cv$$

$$(\gg): ((a \to o) \to o)$$

$$\to (a \to (b \to o) \to o)$$

$$\to (b \to o) \to o$$

$$m \gg k = \lambda c. m(\lambda v. kvc)$$

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

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

$$ashley = \mathbf{a}^{\eta} : \mathcal{C}(e)$$
 (Lexicon)
$$petted = \mathbf{pet}^{\eta} : \mathcal{C}(e \to t)$$

$$every = \lambda P, c. \forall x : Px \to cx : (e \to t) \to \mathcal{C}(e)$$

$$dog = \mathbf{dog} : e \to t$$
 (\$\times \cdot \cdot \cdot (a \to b) \to \mathbb{C}(a) \to \mathbb{C}(b) \tag{Grammar} \tag{Grammar} \\ m \times n = m \simes \lambda f. n \lambda x. (f x)^{\eta} \\ = \lambda c. m(\lambda f. n(\lambda x. c(f x))) \\ (\d\cdot) : \mathbb{C}(a) \to \mathbb{C}(a \to b) \to \mathbb{C}(b) \\ m \lambda n = m \simes \lambda x. n \simes \lambda f. (f x)^{\eta} \\ = \lambda c. m(\lambda x. n(\lambda f. c(f x))) \end{array}

Ashley petted every dog.

 $\texttt{ashley} \triangleleft (\texttt{petted} \triangleright \texttt{everydog})$

$$a^{\eta} \triangleleft (pet^{\eta} \triangleright \text{everydog})$$

Ashley petted every dog.

$$a^\eta \triangleleft (pet^\eta \triangleright \text{every} dog)$$

expand every \mathbf{dog} ...

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

Ashley petted every dog.

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

expand ⊳...

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

Ashley petted every dog.

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

 $expand \mathrel{\triangleleft} \ldots$

$$\lambda c. \forall x : \mathbf{dog} x \to c(\mathbf{pet} x \mathbf{a})$$

Ashley petted every dog.

$$\lambda c. \forall x : \mathbf{dog} x \to c(\mathbf{pet} x \mathbf{a})$$

to obtain a proposition, apply to the identity function...

$$\forall x : \mathbf{dog} x \to \mathbf{pet} x \mathbf{a}$$

Summary

Using continuations to manage scope-taking:

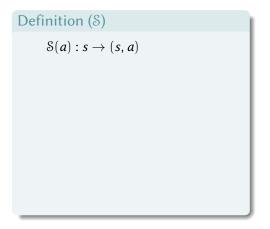
Using continuations to manage scope-taking:

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

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.



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

Definition (S)

$$S(a): s \to (s, a)$$

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

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

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

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

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

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

Definition (S) $S(a): s \to (s, a)$ $(\cdot)^{\eta}: a \to s \to (s, a)$ $v^{\eta} = \lambda s. \langle s, v \rangle$ $(\gg):(s\to(s,a))$ $\rightarrow (a \rightarrow s \rightarrow (s, b))$ $\rightarrow s \rightarrow (s, b)$ $m \gg k = \lambda s. 1 \text{et } \langle s', v \rangle = ms \text{ in } kvs'$

$$\begin{split} \text{ashley} &= \mathbf{a}^{\eta} : \mathcal{S}(e) \\ \text{petted} &= \mathbf{pet}^{\eta} : \mathcal{S}(e \to t) \\ \text{herself} &= \lambda s. \langle s, \text{sel} s \rangle : s \to (s, e) \end{split}$$

$$\begin{aligned} & \text{ashley} = \mathbf{a}^{\eta} : \mathbb{S}(e) & \text{(Lexicon)} \\ & \text{petted} = \mathbf{pet}^{\eta} : \mathbb{S}(e \to t) \\ & \text{herself} = \lambda s. \langle s, \text{sel} s \rangle : \mathbb{S}(e) \end{aligned}$$

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

$$m \triangleright n = m \gg \lambda f. n \gg \lambda x. (fx)^{\eta}$$

$$= \lambda s. \text{let } \langle f, s' \rangle = ms \text{ in let } \langle x, s'' \rangle = ns' \text{ in } \langle fx, s'' \rangle$$

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

$$m \triangleleft n = m \gg \lambda x. n \gg \lambda f. (fx)^{\eta}$$

$$= \lambda s. \text{let } \langle x, s' \rangle = ms \text{ in let } \langle f, s'' \rangle = ns' \text{ in } \langle fx, s'' \rangle$$

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

$$m \triangleright n = m \gg \lambda f. n \gg \lambda x. (fx)^{\eta}$$

$$= \lambda s. \text{let } \langle f, s' \rangle = ms \text{ in let } \langle x, s'' \rangle = ns' \text{ in } \langle fx, s'' \rangle$$

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

$$m \triangleleft n = m \gg \lambda x. n \gg \lambda f. (fx)^{\eta}$$

$$= \lambda s. \text{let } \langle x, s' \rangle = ms \text{ in let } \langle f, s'' \rangle = ns' \text{ in } \langle fx, s'' \rangle$$

$$(\cdot)^{\triangleright} : \mathcal{S}(e) \to \mathcal{S}(e)$$

$$m^{\triangleright} = m \gg \lambda x. s. \langle x :: s. x \rangle$$

Ashley petted herself.

 $ashley^{\triangleright} \triangleleft (petted \triangleright herself)$

$$\texttt{ashley}^{\blacktriangleright} \triangleleft (\textbf{pet}^{\eta} \triangleright \texttt{herself})$$

$$(\lambda s. \langle \mathbf{a} ... s, \mathbf{a} \rangle) \triangleleft (\textbf{pet}^{\eta} \triangleright \texttt{herself})$$

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

Ashley petted herself.

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

expand $\triangleright \dots$

$$(\lambda s. \langle \mathbf{a} :: s, \mathbf{a} \rangle) \triangleleft \lambda s. \langle s, \mathbf{pet}(sels) \rangle$$

Ashley petted herself.

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

expand ⊲...

$$\lambda s. \langle \mathbf{a} :: s, \mathbf{pet}(sels) \mathbf{a} \rangle$$

Using State to manage anaphora:

Using State to manage anaphora:

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

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?

Combining quantification and anaphora

How might one do this?

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

Monad transformers

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

$$\begin{split} & \operatorname{Definition}\left(\mathcal{M}_{\mathcal{T}}\right) \\ & \mathcal{M}_{\mathcal{T}}: (\mathfrak{T} \to \mathfrak{T}) \to \mathfrak{T} \to \mathfrak{T} \\ & (\cdot)^{\mathfrak{n}}: a \to \mathcal{M}_{\mathcal{T}}(\mathcal{M}_{0})(b) \\ & (\gg): \mathcal{M}_{\mathcal{T}}(\mathcal{M}_{0})(a) \to (a \to \mathcal{M}_{\mathcal{T}}(\mathcal{M}_{0})(b)) \to \mathcal{M}_{\mathcal{T}}(\mathcal{M}_{0})(b) \end{split}$$

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...

The Continuation monad transformer

Definition (\mathcal{C}_T)

$$\mathcal{C}_{T}(\mathcal{M}_{0})(a):(a\to\mathcal{M}_{0}(o))\to\mathcal{M}_{0}(o)$$

$$(\cdot)^{\eta}:a\to(a\to\mathcal{M}_{0}(o))\to\mathcal{M}_{0}(o)$$

$$v^{\eta}=\lambda c.cv$$

$$(\ggg):((a\to\mathcal{M}_{0}(o))\to\mathcal{M}_{0}(o))$$

$$\to(a\to(b\to\mathcal{M}_{0}(o))\to\mathcal{M}_{0}(o))$$

$$\to(b\to\mathcal{M}_{0}(o))\to\mathcal{M}_{0}(o)$$

$$m\ggg k=\lambda c.m(\lambda v.kvc)$$

The State monad transformer

Definition (S_T) $S_T(\mathcal{M}_0)(a): s \to \mathcal{M}_0((s,a))$ $(\cdot)^{\eta}: a \to s \to \mathcal{M}_0((s,a))$ $v^{\eta} = \lambda s. \langle s, v \rangle^{\eta}$ $(\gg): (s \to \mathcal{M}_0((s,a)))$ $\rightarrow (a \rightarrow (s \rightarrow \mathcal{M}_0((s,b))))$ $\rightarrow s \rightarrow \mathcal{M}_0((s,b))$ $m \gg k = \lambda s. ms \gg \lambda p. 1 \text{et } \langle s', v \rangle = p \text{ in } kvs'$

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

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?

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?

Every dog licked Ashley. *He is friendly.

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?

Every dog licked Ashley. *He is friendly.

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

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?

Every dog licked Ashley. *He is friendly.

It turns out we must apply C_T to S and not S_T to C.

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

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?

Every dog licked Ashley. *He 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

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?

Every dog licked Ashley. *He 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 $(e \to \mathcal{M}_0(t)) \to \mathcal{M}_0(t) \dots$

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?

Every dog licked Ashley. *He 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 $(e \to \mathcal{M}_0(t)) \to \mathcal{M}_0(t) \dots$
 - but a generic meaning for *every* cannot be written...we are required to know what 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?

Every dog licked Ashley. *He is friendly.

It turns out we must apply C_T to S and not S_T to 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 $(e \to \mathcal{M}_0(t)) \to \mathcal{M}_0(t) \dots$
 - but a generic meaning for *every* cannot be written...we are required to know what 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

- Side effects in linguistic semantics
- Algebraic effects and handlers
- Making it Montagovian
- Quantification and anaphora

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

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

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)

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

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

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

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

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

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)

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

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

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

by characterizing them algebraically

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

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

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...

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)

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

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').

$$E = \{ \operatorname{op}_{1p_1 \leadsto a_1}, \ldots, \operatorname{op}_{np_n \leadsto a_n}, \eta \}$$

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').

$$E = \{ \operatorname{op}_{1p_1 \leadsto a_1}, \ldots, \operatorname{op}_{np_n \leadsto a_n}, \eta \}$$

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').

$$E = \{ \operatorname{op}_{1p_1 \leadsto a_1}, \ldots, \operatorname{op}_{np_n \leadsto a_n}, \eta \}$$

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 $\operatorname{op}_{p \leadsto a}$ is in signature E means that it has the following type signature:

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

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').

$$E = \{ \operatorname{op}_{1p_1 \leadsto a_1}, \ldots, \operatorname{op}_{np_n \leadsto a_n}, \eta \}$$

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 $\operatorname{op}_{p \leadsto a}$ is in signature E means that it has the following type signature:

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

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

$$op_i(p_i; \ldots) = op_j(p_j; \ldots)$$

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

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

two operations, $\operatorname{get}_{\star \leadsto s}$ and $\operatorname{put}_{s \leadsto \star}$

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

• *s* is the type of the state

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

- *s* is the type of the state
- * is the unit type (one inhabitant, also called *)

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

- *s* is the type of the state
- * is the unit type (one inhabitant, also called *)

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

- *s* is the type of the state
- ★ is the unit type (one inhabitant, also called ★)

Some example elements of the State algebra...

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

- *s* is the type of the state
- * is the unit type (one inhabitant, also called *)

Some example elements of the State algebra...

• $get_{\star \leadsto s}(\star; \lambda s.\eta s) : \mathcal{F}_{\{get_{\star \leadsto s}, put_{s \leadsto \star}\}}(s)$

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

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

Some example elements of the State algebra...

- \bullet get_{*\inftys}(*; \lambda s.\eta s): \mathcal{F}_{\{get_{*\inftys}}, \text{put}_{s\inftys}}(s)
- $\bullet \ \operatorname{get}_{\star \leadsto s}(\star; \lambda s.\operatorname{put}_{s \leadsto \star}(\mathbf{a} :: s; \lambda \star . \eta s)) : \mathcal{F}_{\{\operatorname{get}_{\star \leadsto s}, \operatorname{put}_{s \leadsto \star}\}}(s)$

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

two operations, $get_{\star \leadsto s}$ and $put_{s \leadsto \star}$

- *s* is the type of the state
- ★ is the unit type (one inhabitant, also called ★)

Some example elements of the State algebra...

- $get_{\star \leadsto s}(\star; \lambda s.\eta s) : \mathcal{F}_{\{get_{\star \leadsto s}, put_{s \leadsto \star}\}}(s)$
- $\bullet \ \operatorname{get}_{\star \leadsto s}(\star; \lambda s.\operatorname{put}_{s \leadsto \star}(\mathbf{a} :: s; \lambda \star .\eta s)) : \mathcal{F}_{\{\operatorname{get}_{\star \leadsto s}, \operatorname{put}_{s \leadsto \star}\}}(s)$
- $\bullet \ \operatorname{get}_{\star \leadsto s}(\star; \lambda s.\operatorname{put}_{s \leadsto \star}(\mathbf{a} :: s; \lambda \star .\eta \mathbf{a})) : \mathcal{F}_{\{\operatorname{get}_{\star \leadsto s}, \operatorname{put}_{s \leadsto \star}\}}(e)$

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

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

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

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

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

$$get_{\star \leadsto s}(\star; \lambda g.put_{s \leadsto \star}(g; k))) = k \star$$

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

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

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

$$get_{\star \leadsto s}(\star; \lambda g.put_{s \leadsto \star}(g; k))) = k \star$$

Getting something that you just put back is the same as doing nothing:

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

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

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

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

$$get_{\star \leadsto s}(\star; \lambda g.put_{s \leadsto \star}(g; k))) = k \star$$

Getting something that you just put back is the same as doing nothing:

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

Putting twice overwrites:

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

The Quantifier algebra (signature)

one operation, $scope_{(e \rightarrow t) \rightarrow t \leadsto e}$

The Quantifier algebra (signature)

one operation, $scope_{(e \rightarrow t) \rightarrow t \leadsto e}$

An example element of the Quantifier algebra...

 $\bullet \ \operatorname{scope}_{(e \to t) \to t \leadsto e}((\lambda k. \forall x : \operatorname{dog} x \to kx); \lambda y. \eta(\operatorname{sleep} y)) : \mathcal{F}_{\{\operatorname{scope}_{(e \to t) \to t \leadsto e}\}(t)}$

The Quantifier algebra (laws)

Quantifying in:

$$scope_{(e \to t) \to t \to e}(q; \lambda x. \eta(kx)) = \eta(qk)$$

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

We are here

- Side effects in linguistic semantics
- 2 Algebraic effects and handlers
- Making it Montagovian
- Quantification and anaphora

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.

Encoding operations

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

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

Encoding operations

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

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

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

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

We are here

- Side effects in linguistic semantics
- 2 Algebraic effects and handlers
- Making it Montagovian
- Quantification and anaphora

- 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.

- 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.

- 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.

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.

- 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.