Pragmatic Side Effects

Jirka Maršík and Maxime Amblard

LORIA, UMR 7503, Université de Lorraine, CNRS, Inria, Campus Scientifique,
F-54506 Vandœuvre-lès-Nancy, France

20 March, 2015

Our Setting

Context:

- Montague semantics, using the λ calculus

Objective:

- Increase the empirical coverage

Challenge:

- multiple sentences
  - discourse phenomena
  - pragmatics

Example of Pragmasemantics

de Groote – Type-Theoretic Dynamic Logic

Montague

$$\llbracket s \rrbracket = o$$
$$\llbracket n \rrbracket = \iota \to \llbracket s \rrbracket$$
$$\llbracket np \rrbracket = (\iota \to \llbracket s \rrbracket) \to \llbracket s \rrbracket$$

de Groote

$$\llbracket s \rrbracket = \gamma \to (\gamma \to o) \to o$$
$$\llbracket n \rrbracket = \iota \to \llbracket s \rrbracket$$
$$\llbracket np \rrbracket = (\iota \to \llbracket s \rrbracket) \to \llbracket s \rrbracket$$

$$\llbracket \textit{He bought a car} \rrbracket = \lambda e \phi.\, \exists x. \mathit{car}(x) \land \mathit{bought}(\mathtt{sel}_{he}(e), x) \land \phi(x{::}e)$$

Example of Pragmasemantics

de Groote – Type-Theoretic Dynamic Logic

Montague

$$\llbracket s \rrbracket = o$$
$$\llbracket n \rrbracket = \iota \to \llbracket s \rrbracket$$
$$\llbracket np \rrbracket = (\iota \to \llbracket s \rrbracket) \to \llbracket s \rrbracket$$

de Groote

$$\llbracket s \rrbracket = \gamma \to (\gamma \to o) \to o$$
$$\llbracket n \rrbracket = \iota \to \llbracket s \rrbracket$$
$$\llbracket np \rrbracket = (\iota \to \llbracket s \rrbracket) \to \llbracket s \rrbracket$$

$$\llbracket \textit{He bought a car} \rrbracket = \lambda e \phi.\, \exists x. car(x) \wedge bought(\mathtt{sel}_{he}(e), x) \wedge \phi(x{::}e)$$

Drawing Inspiration from Programming Languages

There is in my opinion no important theoretical difference between natural languages and the programming languages of computer scientists.

Side Effects in Programming Languages

Account for:

- a program's interaction with the world of its users
  - e.g., makings sounds, printing documents, moving robotic limbs...
- non-local interactions between parts of a program
  - e.g., writing to and reading from variables, throwing and catching exceptions...

Side effects align with pragmatics in their purpose.

Side Effects in Programming Languages

Account for:

- a program's interaction with the world of its users
  - e.g., makings sounds, printing documents, moving robotic limbs...
- non-local interactions between parts of a program
  - e.g., writing to and reading from variables, throwing and catching exceptions...

Side effects align with pragmatics in their purpose.

Side Effects in Programming Languages

Account for:

- a program's interaction with the world of its users
  - e.g., makings sounds, printing documents, moving robotic limbs...
- non-local interactions between parts of a program
  - e.g., writing to and reading from variables, throwing and catching exceptions...

Side effects align with pragmatics in their purpose.

Side Effects in Programming Languages

Account for:

- a program's interaction with the world of its users
  - e.g., makings sounds, printing documents, moving robotic limbs...
- non-local interactions between parts of a program
  - e.g., writing to and reading from variables, throwing and catching exceptions...

Side effects align with pragmatics in their purpose.

Type Raising

Side effects and pragmatics align also in their theories.

Most famous example: Montague's type raising

- from entities to generalized quantifiers
- i.e., from ι to $(\iota \to o) \to o$
- e.g., *john* becomes $\lambda P.P\ john$

In computer science, discovered as continuations

- raising α to $(\alpha \to \omega) \to \omega$
- e.g., applying a function f to two arguments S and O in continuation-passing style

$$\lambda P.S(\lambda x.O(\lambda y.P\ (f\ x\ y)))$$

Type Raising

Side effects and pragmatics align also in their theories.

Most famous example: Montague's type raising

- from entities to generalized quantifiers
- i.e., from ι to $(\iota \to o) \to o$
- e.g., *john* becomes $\lambda P.P\ john$

In computer science, discovered as continuations

- raising α to $(\alpha \to \omega) \to \omega$
- e.g., applying a function f to two arguments S and O in continuation-passing style

$$\lambda P.S(\lambda x.O(\lambda y.P\ (f\ x\ y)))$$

Type Raising

Side effects and pragmatics align also in their theories.

Most famous example: Montague's type raising

- from entities to generalized quantifiers
- i.e., from ι to $(\iota \to o) \to o$
- e.g., *john* becomes $\lambda P.P\ john$

In computer science, discovered as continuations

- raising α to $(\alpha \to \omega) \to \omega$
- e.g., applying a function f to two arguments S and O in continuation-passing style

$$\lambda P.S(\lambda x.O(\lambda y.P\ (f\ x\ y)))$$

Generalizing Denotations

"Upgrading" the types of denotations in order to keep a compositional semantics seems like a common strategy.

Natural Languages	Prog. Languages	Type α becomes
Quantification	Control	$(\alpha \to \omega) \to \omega$
Anaphora	State	$\gamma \to \alpha \times \gamma$
Intensionality	Environment	$\delta \to \alpha$
Presuppositions	Exceptions	$\alpha \oplus \chi$
Questions	Non-determinism	$\alpha \to o$
Focus		$\alpha \times (\alpha \to o)$
Expressives	Output	$\alpha \times \epsilon$
Prob. semantics	Prob. programming	$[\mathbb{R} \times \alpha]$

How to Avoid Changing Denotations?

Different pragmasemantic phenomena, all in one theory
$\rightarrow$ more and more elaborate types

We often have to change our minds on what is meaning

- old denotations $\rightarrow$ outdated
- denotations from other strands of work $\rightarrow$ incompatible

Some solutions to this problem exist already in computer science.

How to Avoid Changing Denotations?

Different pragmasemantic phenomena, all in one theory
$\rightarrow$ more and more elaborate types

We often have to change our minds on what is meaning

- old denotations $\rightarrow$ outdated
- denotations from other strands of work $\rightarrow$ incompatible

Some solutions to this problem exist already in computer science.

How to Avoid Changing Denotations?

Different pragmasemantic phenomena, all in one theory
$\rightarrow$ more and more elaborate types

We often have to change our minds on what is meaning

- old denotations $\rightarrow$ outdated
- denotations from other strands of work $\rightarrow$ incompatible

Some solutions to this problem exist already in computer science.

Example from Computer Science

3		
3		

Example from Computer Science

3	**x + 3**	
3		
$\lambda s. \langle 3, s\rangle$	$\lambda s. \langle s(''x'') + 3, s\rangle$	

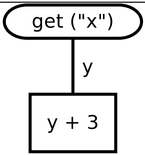
Example from Computer Science

3	**x + 3**	**print("hello")**
3		
$\lambda s.\langle 3, s\rangle$	$\lambda s.\langle s(''x'') + 3, s\rangle$	
$\lambda s.\langle 3, s, ''''\rangle$	$\lambda s.\langle s(''x'') + 3, s, ''''\rangle$	$\lambda s.\langle (), s, ''hello''\rangle$

Example from Computer Science

3	**x + 3**	**print("hello")**
3		
$\lambda s.\langle 3, s\rangle$	$\lambda s.\langle s("x") + 3, s\rangle$	
$\lambda s.\langle 3, s, ""\rangle$	$\lambda s.\langle s("x") + 3, s, ""\rangle$	$\lambda s.\langle (), s, "hello"\rangle$
$\boxed{3}$		

Example from Computer Science

3	**x + 3**	**print("hello")**
3		
$\lambda s.\,\langle 3, s\rangle$	$\lambda s.\,\langle s(\text{"}x\text{"}) + 3, s\rangle$	
$\lambda s.\,\langle 3, s, \text{""}\rangle$	$\lambda s.\,\langle s(\text{"}x\text{"}) + 3, s, \text{""}\rangle$	$\lambda s.\,\langle (), s, \text{"}hello\text{"}\rangle$
$\boxed{3}$		
$\boxed{3}$		

Example from Computer Science

3	**x + 3**	**print("hello")**
3		
$\lambda s.\,\langle 3, s\rangle$	$\lambda s.\,\langle s(\text{"}x\text{"}) + 3, s\rangle$	
$\lambda s.\,\langle 3, s, \text{""}\rangle$	$\lambda s.\,\langle s(\text{"}x\text{"}) + 3, s, \text{""}\rangle$	$\lambda s.\,\langle (), s, \text{"}hello\text{"}\rangle$
$\boxed{3}$		
$\boxed{3}$	get ("x") $\xrightarrow{y}$ $\boxed{y + 3}$	
$\boxed{3}$	get ("x") $\xrightarrow{y}$ $\boxed{y + 3}$	print ("hello") $\xrightarrow{()}$ $\boxed{()}$

Example from Linguistics

John		
j		

Example from Linguistics

John	**every boy**	
j		
$\lambda P.Pj$	$\lambda P.\forall x.(boy\ x) \to (P\ x)$	

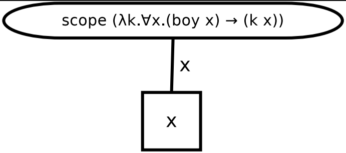
Example from Linguistics

John	**every boy**	**she**
j		
$\lambda P.Pj$	$\lambda P.\forall x.(boy\ x) \to (P\ x)$	
$\lambda Pe\phi.Pj(j :: e)\phi$	$\lambda Pe\phi.[\forall x.(boy\ x) \to$ $P\ x\ (x :: e)\ (\lambda e'.\top)] \wedge \phi\ e$	$\lambda Pe\phi.P(\mathtt{sel}_{she}(e))e\phi$

Example from Linguistics

John	**every boy**	**she**
j		
$\lambda P.Pj$	$\lambda P.\forall x.(boy\ x) \rightarrow (P\ x)$	
$\lambda Pe\phi.Pj(j :: e)\phi$	$\lambda Pe\phi.[\forall x.(boy\ x) \rightarrow$ $P\ x\ (x :: e)\ (\lambda e'.\top)] \wedge \phi\ e$	$\lambda Pe\phi.P(\mathtt{sel}_{she}(e))e\phi$
$\boxed{\text{j}}$		

Example from Linguistics

John	**every boy**	**she**
j		
$\lambda P.Pj$	$\lambda P.\forall x.(boy\ x) \rightarrow (P\ x)$	
$\lambda Pe\phi.Pj(j :: e)\phi$	$\lambda Pe\phi.[\forall x.(boy\ x) \rightarrow$ $P\ x\ (x :: e)\ (\lambda e'.\top)] \wedge \phi\ e$	$\lambda Pe\phi.P(\mathtt{sel}_{she}(e))e\phi$
$\boxed{j}$		
$\boxed{j}$		

Example from Linguistics

John	**every boy**	**she**
j		
$\lambda P.Pj$	$\lambda P.\forall x.(boy\ x) \to (P\ x)$	
$\lambda Pe\phi.Pj(j :: e)\phi$	$\lambda Pe\phi.[\forall x.(boy\ x) \to P\ x\ (x :: e)\ (\lambda e'.\top)] \wedge \phi\ e$	$\lambda Pe\phi.P(\mathtt{sel}_{she}(e))e\phi$
$\boxed{\text{j}}$		
$\boxed{\text{j}}$	$\text{scope } (\lambda k.\forall x.(\text{boy } x) \to (k\ x)) \xrightarrow{x} \boxed{\text{x}}$	
$\text{push (j)} \xrightarrow{()} \boxed{\text{j}}$	$\text{scope } (\lambda k.\forall^{d} x.[(\text{boy } x) \to (k\ x)]) \xrightarrow{x} \boxed{\text{x}}$	$\text{get ()} \xrightarrow{e} \boxed{\text{sel\_she(e)}}$

Reaping the Benefits of Stability

Consider the semantics of a relational noun like *mother* in the construction *the mother of X*.

$$[\![\textit{the mother of}]\!] = \lambda x.\ mother(x)$$
$$[\![\textit{the mother of}]\!] = \lambda XP.\ X\ (\lambda x.\ P\ (mother(x)))$$
$$[\![\textit{the mother of}]\!] = \lambda XPe\phi.\ X\ (\lambda xe'\phi'.\ P\ (mother(x))\ e'\ \phi')\ e\ \phi$$

Its denotation changes even though the meaning stays morally the same.

Reaping the Benefits of Stability

Consider the semantics of a relational noun like *mother* in the construction *the mother of X*.

$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$

$$[\![\textit{the mother of}]\!] = \lambda X P.\ X\ (\lambda x.\ P\ (\textit{mother}(x)))$$

$$[\![\textit{the mother of}]\!] = \lambda X P e \phi.\ X\ (\lambda x e' \phi'.\ P\ (\textit{mother}(x))\ e'\ \phi')\ e\ \phi$$

Its denotation changes even though the meaning stays morally the same.

Reaping the Benefits of Stability

Consider the semantics of a relational noun like *mother* in the construction *the mother of X*.

$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$
$$[\![\textit{the mother of}]\!] = \lambda XP.\ X\ (\lambda x.\ P\ (\textit{mother}(x)))$$
$$[\![\textit{the mother of}]\!] = \lambda XPe\phi.\ X\ (\lambda xe'\phi'.\ P\ (\textit{mother}(x))\ e'\ \phi')\ e\ \phi$$

Its denotation changes even though the meaning stays morally the same.

Reaping the Benefits of Stability

Consider the semantics of a relational noun like *mother* in the construction *the mother of X*.

$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$
$$[\![\textit{the mother of}]\!] = \lambda XP.\ X\ (\lambda x.\ P\ (\textit{mother}(x)))$$
$$[\![\textit{the mother of}]\!] = \lambda XPe\phi.\ X\ (\lambda x e' \phi'.\ P\ (\textit{mother}(x))\ e'\ \phi')\ e\ \phi$$

Its denotation changes even though the meaning stays morally the same.

Reaping the Benefits of Stability

Consider the semantics of a relational noun like *mother* in the construction *the mother of X*.

$$[\![\textit{the mother of}]\!] = \lambda x.\ mother(x)$$
$$[\![\textit{the mother of}]\!] = \lambda XP.\ X\ (\lambda x.\ P\ (mother(x)))$$
$$[\![\textit{the mother of}]\!] = \lambda XPe\phi.\ X\ (\lambda xe'\phi'.\ P\ (mother(x))\ e'\ \phi')\ e\ \phi$$

Its denotation changes even though the meaning stays morally the same.

Reaping the Benefits of Stability

How does it work in our system?

$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$

Reaping the Benefits of Stability

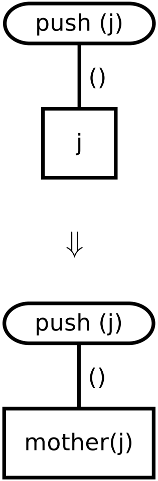
How does it work in our system?

$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$

Reaping the Benefits of Stability

How does it work in our system?

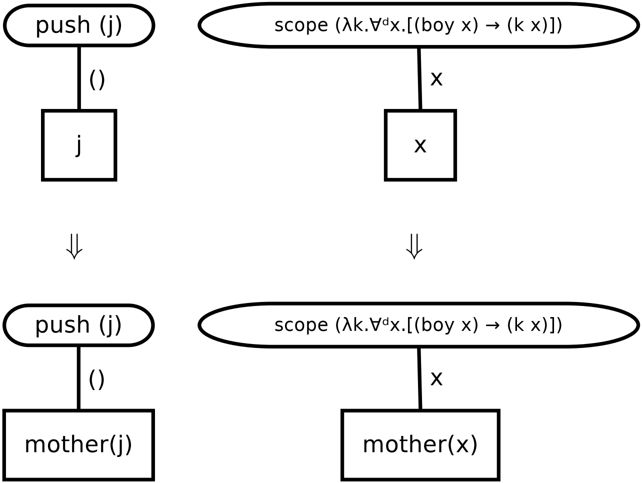
$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$



Reaping the Benefits of Stability

How does it work in our system?

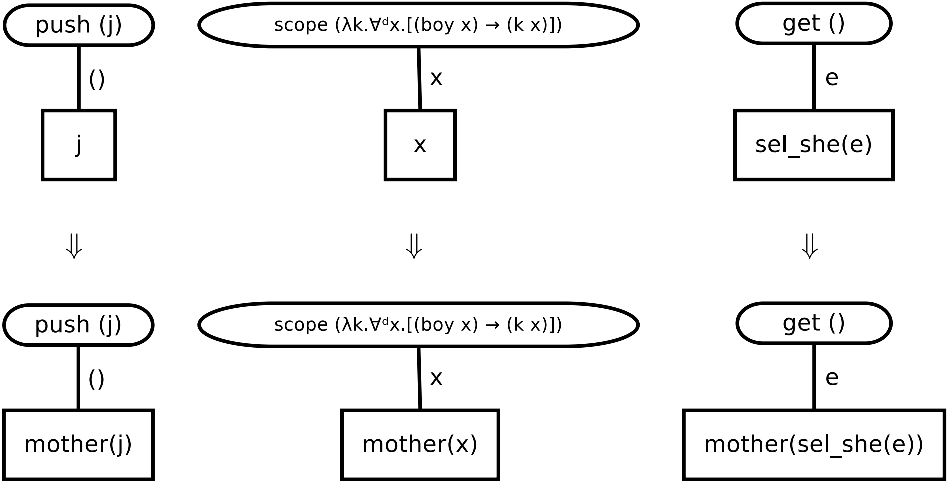
$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$



Reaping the Benefits of Stability

How does it work in our system?

$$[\![\textit{the mother of}]\!] = \lambda x.\ \textit{mother}(x)$$



Reaping the Benefits of Stability

Our meaning for *the mother of X* is agnostic about its argument.
It works with simple, quantificational or dynamic meanings of X.

This also holds for more involved meanings of the relational noun.

Reaping the Benefits of Stability

Our meaning for *the mother of X* is agnostic about its argument. It works with simple, quantificational or dynamic meanings of X.

This also holds for more involved meanings of the relational noun.

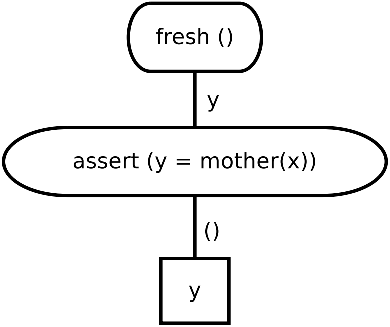
Reaping the Benefits of Stability

Our meaning for *the mother of X* is agnostic about its argument.
It works with simple, quantificational or dynamic meanings of X.

This also holds for more involved meanings of the relational noun.

- dynamic *mother*

$[\![\textit{the mother of}]\!] = \lambda x.$



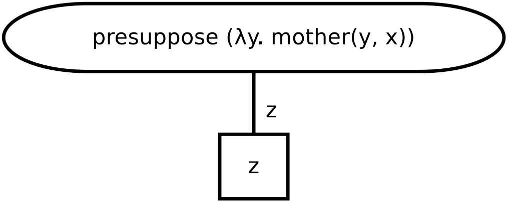
Reaping the Benefits of Stability

Our meaning for *the mother of X* is agnostic about its argument. It works with simple, quantificational or dynamic meanings of *X*.

This also holds for more involved meanings of the relational noun.

- presuppositional *mother*

$$[\![\textit{the mother of}]\!] = \lambda x.$$



Algebraic Effects...

We have been using a framework developed in PL research.

In it:

- interacting with the context = throwing an exception
- the exception contains a response for every possible outcome of the operation

Denotations are:

- algebraic expressions (drawn as trees)
- generators = values
- operators = possible interactions with the context
- arity = the number of possible outcomes
- type = $\mathcal{F}_{\Sigma}(\tau)$

Algebraic Effects. . .

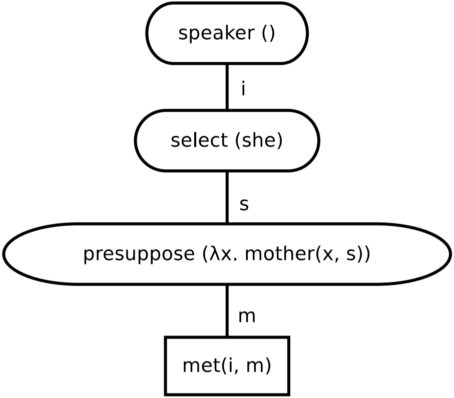
We have been using a framework developed in PL research.

In it:

- interacting with the context = throwing an exception
- the exception contains a response for every possible outcome of the operation

Denotations are:

- algebraic expressions (drawn as trees)
- generators = values
- operators = possible interactions with the context
- arity = the number of possible outcomes
- type = $\mathcal{F}_{\Sigma}(\tau)$



. . . and Handlers

Handlers give scope and interpretation to (some of) the effects in a computation.

- Practically, they are like exception handlers in programming languages.
- Technically, they are catamorphisms (folds) on the algebra of effects.

Examples:

- a tensed verb delimits quantification, creating a scope island
- logical negation blocks referent accessibility (as in DRT or TTDL)
- the common ground accomodates presuppositions if they have not been yet assumed
- hypotheseses can cancel presuppositions in their scope (if . . . , then . . .)

. . . and Handlers

Handlers give scope and interpretation to (some of) the effects in a computation.

- Practically, they are like exception handlers in programming languages.
- Technically, they are catamorphisms (folds) on the algebra of effects.

Examples:

- a tensed verb delimits quantification, creating a scope island
- logical negation blocks referent accessibility (as in DRT or TTDL)
- the common ground accomodates presuppositions if they have not been yet assumed
- hypotheseses can cancel presuppositions in their scope (if . . . , then . . .)

Proof of Concept

We have built a small prototype to test and explore our approach.

- in-situ quantification
- discourse anaphora
- presuppositions (of referentials)
- their interactions
  - e.g., binding problem

Summary

- perspective shift
  - from denotations as complex objects
    to denotations as complex processes producing simple objects
  - focus on what meanings do, not on what they are
- content/context distinction
  - objects – purely truth-conditional material
  - process – we dump the pragmatic wastebasket here
  - placement of non-locality phenomena such as in-situ quantification is to our discretion
- easier to manage multiple effects
  - our driving motivation (empirical coverage)
  - stable denotations help avoid generalizing to the worst case
  - captures parameters, mutable state, continuations, projections and their filtering/cancelling both flexibly and compositionally
    - used in PLT research and functional programming too

Summary

- perspective shift
  - from denotations as complex objects
    to denotations as complex processes producing simple objects
  - focus on what meanings do, not on what they are
- content/context distinction
  - objects – purely truth-conditional material
  - process – we dump the pragmatic wastebasket here
  - placement of non-locality phenomena such as in-situ quantification is to our discretion
- easier to manage multiple effects
  - our driving motivation (empirical coverage)
  - stable denotations help avoid generalizing to the worst case
  - captures parameters, mutable state, continuations, projections and their filtering/cancelling both flexibly and compositionally
    - used in PLT research and functional programming too

Summary

- perspective shift
  - from denotations as complex objects
    to denotations as complex processes producing simple objects
  - focus on what meanings do, not on what they are
- content/context distinction
  - objects – purely truth-conditional material
  - process – we dump the pragmatic wastebasket here
  - placement of non-locality phenomena such as in-situ quantification is to our discretion
- easier to manage multiple effects
  - our driving motivation (empirical coverage)
  - stable denotations help avoid generalizing to the worst case
  - captures parameters, mutable state, continuations, projections and their filtering/cancelling both flexibly and compositionally
    - used in PLT research and functional programming too