

# Syntax-semantics interface and the non-trivial computation of meaning

APA/ASL Group Meeting GVI-2: Lambda Calculi, Type Systems, and  
Applications to Natural Language

APA Eastern Division 108th Annual Meeting  
Washington, DC, December 28, 2011

We describe the relationship between the surface form of a sentence and its meaning as two non-trivial but mechanical interpretations of the same (hidden) abstract form. Our approach, in the tradition of abstract categorial grammar (ACG), deals with several languages: The language  $\mathcal{A}$  is the language of abstract form; the language  $\mathcal{I}$  is the language in which we write a compositional interpreter of  $\mathcal{A}$ , which produces a term in a language  $\mathcal{T}$ . The language  $\mathcal{A}$  is simply-typed lambda-calculus with constants.

A syntactic interpretation  $\mathcal{I}_{\text{syn}}$  gives a surface form for a sentence corresponding to  $\mathcal{A}$ ;  $\mathcal{T}_{\text{syn}}$  is a set of strings or utterances. This interpretation is responsible for word order, case marking, subject-verb agreement, etc. Therefore,  $\mathcal{A}$  is abstracted from these syntactic details.  $\mathcal{A}$  can be determined from  $\mathcal{T}_{\text{syn}}$  by parsing. If the abstract language is simple – that is, its constants have types of low order – parsing is tractable. This is the case for our ACGs.

A semantic interpretation  $\mathcal{I}_{\text{sem}}$  computes the meaning of  $\mathcal{A}$  as a logical formula;  $\mathcal{T}_{\text{sem}}$  is thus the language of higher-order logic.  $\mathcal{I}_{\text{sem}}$  deals with quantifiers, pronouns, question words and their interactions. Our  $\mathcal{I}_{\text{sem}}$  interpreter is non-trivial: it may fail. Although we may parse a sentence to an abstract form, we may fail to compute any meaning for it, if the sentence just ‘doesn’t make sense’. Our  $\mathcal{I}_{\text{sem}}$  is not a regular lambda-calculus: it expresses computational effects such as mutation and continuations, and it is not normalizable.

# Outline

## ► Overview

Details

Demonstration of the syntax-semantics calculator

# Syntax-semantics interface

Syntax



Semantics

Goal: investigate the correspondence between the syntax and semantics

# Syntax-semantics interface

Syntax  $\longleftrightarrow$  Semantics  
"John ignored a woman"  $\longleftrightarrow \exists_j (\textit{woman } j) \wedge (\textit{ignore john } j)$

What does it mean. Syntax: the surface form such as plain text or an utterance. Semantics: logical formula in first-order predicate logic.

# Syntax-semantics interface

Abstract

fullstop \*(ignored \*(a \*woman) \*John)

$\mathcal{I}_{\text{syn}} \downarrow$

$\downarrow \mathcal{I}_{\text{sem}}$

Syntax

"John ignored a woman"

Semantics

$\exists_j (\textit{woman } j) \wedge (\textit{ignore } \textit{john } j)$

We follow the Abstract Categorical Grammar approach: both syntax and semantics are the transformations from a common, abstract language

# Syntax-semantics interface

Abstract

fullstop \*(ignored \*(a \*woman) \*John)

Syntax

"John ignored a woman"

Semantics

$\exists_j (woman\ j) \wedge (ignore\ john\ j)$

What is a language?

What is a language: here are the examples. We will define formally later, for now: set of tokens such as strings or terms connected with binary operators and generated by a particular grammar.

# Syntax-semantics interface

Abstract

fullstop \*(ignored \*(a \*woman) \*John)

$\mathcal{I}_{\text{syn}}'$

$\mathcal{I}_{\text{syn}} \downarrow$

$\downarrow \mathcal{I}_{\text{sem}}$

$\mathcal{I}_{\text{sem}}'$

Syntax

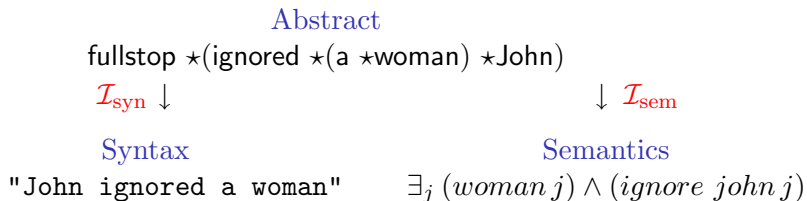
"John ignored a woman"

Semantics

$\exists_j (\textit{woman } j) \wedge (\textit{ignore } \textit{john } j)$

The abstract language can be transformed to several concrete languages; there may also be alternative semantics transformations.

# Syntax-semantics interface



What is a transformation?

What are transformations? They are specified in some language, for example, English. We aim at the formal specification of transformations, expressed in a formal language with precise evaluation rules – like  $\lambda$ -calculus.

# Syntax-semantics interface

Abstract

fullstop \*(ignored \*(a \*woman) \*John)

$\mathcal{I}_{\text{syn}}$   $\uparrow$

$\downarrow$   $\mathcal{I}_{\text{sem}}$

Syntax

"John ignored a woman"

Semantics

$\exists_j (\textit{woman } j) \wedge (\textit{ignore john } j)$

Parsing

Grammaticality and parsing. A sentence in the concrete syntax is ungrammatical if there is no corresponding abstract sentence: that is, it cannot be parsed.

# Syntax-semantics interface

Abstract

fullstop \*(ignored \*(a \*woman) \*John)

$\mathcal{I}_{\text{syn}} \downarrow$

$\downarrow \mathcal{I}_{\text{sem}}$

Syntax

"John ignored a woman"

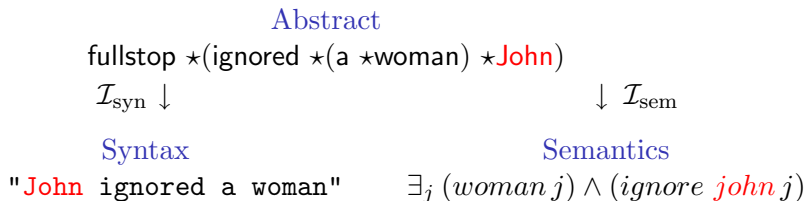
Semantics

$\exists_j (\textit{woman } j) \wedge (\textit{ignore } \textit{john } j)$

$\mathcal{I}_{\text{sem}}$  is non-trivial: it may fail

Also, although a sentence may parse to an abstract form,  $\mathcal{I}_{\text{sem}}$  may fail to produce the logical formula. A sentence may just make no sense.

# Syntax-semantics interface



$\mathcal{I}$  are composable and *modular*

Transformations should be modular and composable: the transformation of **John** to "**John**" and *john* which is written for this sentence should work for all other sentences with John.

# Syntax-semantics interface

## Abstract

fullstop \*(ignored \*(a \*woman) \*John)

$\mathcal{I}_{\text{syn}} \downarrow$

$\downarrow \mathcal{I}_{\text{sem}}$

$\mathcal{I}_{\text{syn}_1} \downarrow$

$\downarrow \mathcal{I}_{\text{sem}_1}$

Syntax

Semantics

"John ignored a woman"

$\exists_j (\textit{woman } j) \wedge (\textit{ignore john } j)$

$\mathcal{I}$  are composable from smaller features

Transformations compose in a different sense: the whole  $\mathcal{I}_{\text{syn}}$  or  $\mathcal{I}_{\text{sem}}$  can be built from smaller, independent steps. We'll see the examples of that later.

## Sample sentences

- (1) John ignored Mary and she left.
- (2) John ignored a woman and she left.
- (3) John ignored every woman and she left.

...

- ▶ binding

BTW, here some of the example sentences for this talk, to illustrate the range of handled linguistic phenomena. Binding, (1), (2), (3).

## Sample sentences

- (1) John ignored Mary and she left.
- (2) John ignored a woman and she left.
- (3) John ignored every woman and she left.

...

- ▶ binding
- ▶ quantification and binding

A quantifier can bind a variable within its scope, (2).

## Sample sentences

- (1) John ignored Mary and she left.
- (2) John ignored a woman and she left.
- (3) John ignored every woman and she left.

...

- ▶ binding
- ▶ quantification and binding
- ▶ different scope of different quantifiers

Different quantifiers have different scope abilities. Universals are clause- or sentence-bound, (3), but indefinites can scope out of a clause or a sentence (2).

# Outline

Overview

► **Details**

Demonstration of the syntax-semantics calculator

That was the overview of the whole talk. Now we get to the details. We start with Greek, and then move to a less flowery notation, which can be mechanically manipulated.

## Running example

John ignored a woman.

Now we give the details of the ACG and precise definitions.

# Abstract signature

## A higher-order signature

A collection of atomic types, constants, and type assignments to constants

Signature  $\Sigma_{\text{abs}}$

Atomic types

$CN, NP, S, M$

John

$: NP$

woman

$: CN$

a

$: CN \rightsquigarrow NP$

ignored

$: NP \rightsquigarrow NP \rightsquigarrow S$

fullstop

$: S \rightsquigarrow M$

Composition (elimination rule for  $\rightsquigarrow$ )

★

$: (a \rightsquigarrow b) \rightarrow a \rightarrow b$

Language  $\stackrel{\text{def}}{=} \text{the set of the typed terms you can build from the signature}$

Abstract, semantic, etc. languages are defined the same way, by a signature, which enumerates types and constants, and specifies the single composition operation,  $\star$ .

# Abstract signature

## A higher-order signature

A collection of atomic types, constants, and type assignments to constants

Signature  $\Sigma_{\text{abs}}$

Atomic types

$CN, NP, S, M$

John

$: NP$

woman

$: CN$

a

$: CN \rightsquigarrow NP$

ignored

$: NP \rightsquigarrow NP \rightsquigarrow S$

fullstop

$: S \rightsquigarrow M$

Composition (elimination rule for  $\rightsquigarrow$ )

★

$: (a \rightsquigarrow b) \rightarrow a \rightarrow b$

Language  $\stackrel{\text{def}}{=}$  the set of the typed terms you can build from the signature

The only uncommon parts here are the type  $M$ , for the complete matrix sentence or discourse, and 'fullstop', the end of the discourse (or sentence, in this case). Also unlike ACG, we introduce type constructor  $\rightsquigarrow$  for the abstract function space. One may regard  $\star$  as the elimination rule for  $\rightsquigarrow$ .

# Abstract terms

Language  $\stackrel{\text{def}}{=} \text{the set of the typed terms you can build from the signature}$

Terms over  $\Sigma_{\text{abs}}$

$e ::= c \mid e \star e, \quad c \in \Sigma_{\text{abs}}$

A sample term

$t_{\text{woman}} \stackrel{\text{def}}{=} \text{fullstop (ignored (a woman) John)}$

One can verify that the term  $t_{woman}$  is well-typed and so it is in the set of typed lambda-terms over the abstract signature.

# String signature

Signature  $\Sigma_{\text{str}}$

Atomic type    string

"John"        : string

"woman"      : string

"a"            : string

"ignored"    : string

". "          : string

Composition

◇              : string  $\rightarrow$  string  $\rightarrow$  string

The operation  $\diamond$  denotes string concatenation. It looks like a simple version of the signature.

$\mathcal{I}_{\text{syn}}$ 

$\mathcal{I}_{\text{syn}}$ : mapping of constants of  $\Sigma_{\text{abs}}$  to terms over  $\Sigma_{\text{str}}$   
 $\lambda$ -calculus is the meta-language to define  $\mathcal{I}_{\text{syn}}$

N, NP, S, and M	$\mapsto$	string
$\rightsquigarrow$	$\mapsto$	$\rightarrow$
John	$\mapsto$	"John"
woman	$\mapsto$	"woman"
a	$\mapsto$	"a"
ignored	$\mapsto$	$\lambda o. \lambda s. s \diamond \text{"ignored"} \diamond o$
fullstop	$\mapsto$	$\lambda x. x \diamond \text{"."}$
*	$\mapsto$	function application

$\mathcal{I}_{\text{syn}}$  is the eval. It *interprets* constants of the abstract signature in, here, the surface language. The interpretation of constants homomorphically extends to the interpretation of the whole abstract language in terms of the surface language.

$\mathcal{I}_{\text{syn}}$ 

$\mathcal{I}_{\text{syn}}$ : mapping of constants of  $\Sigma_{\text{abs}}$  to terms over  $\Sigma_{\text{str}}$   
 $\lambda$ -calculus is the meta-language to define  $\mathcal{I}_{\text{syn}}$

N, NP, S, and M	$\mapsto$	string
$\rightsquigarrow$	$\mapsto$	$\rightarrow$
John	$\mapsto$	"John"
woman	$\mapsto$	"woman"
a	$\mapsto$	"a"
ignored	$\mapsto$	$\lambda o. \lambda s. s \diamond \text{"ignored"} \diamond o$
fullstop	$\mapsto$	$\lambda x. x \diamond \text{"."}$
$\star$	$\mapsto$	function application

the surface form

$$\begin{aligned}
 & \mathcal{I}_{\text{syn}}(t_{\text{woman}}) \\
 = & (\lambda x. x \diamond \text{"."})((\lambda o. \lambda s. s \diamond \text{"ignored"} \diamond o) \\
 & ((\lambda x. \text{"a"} \diamond x) \text{"woman"}) \text{"John"}) \\
 \hookrightarrow & \text{"John"} \diamond \text{"ignored"} \diamond \text{"a"} \diamond \text{"woman"} \diamond \text{"."}
 \end{aligned}$$

I must emphasize a point that becomes very important later. If we just substitute for the constants in the sample woman term their mapped terms, we get this long phrase on the second line in the table. The whole transformation is trivial: simple exchange taking care of the word order. We can implement case marking, gender and number agreement, etc.

$\mathcal{I}_{\text{syn}}$ : mapping of constants of  $\Sigma_{\text{abs}}$  to terms over  $\Sigma_{\text{str}}$   
 $\lambda$ -calculus is the meta-language to define  $\mathcal{I}_{\text{syn}}$

N, NP, S, and M	$\mapsto$	string
$\rightsquigarrow$	$\mapsto$	$\rightarrow$
John	$\mapsto$	"John"
woman	$\mapsto$	"woman"
a	$\mapsto$	"a"
ignored	$\mapsto$	$\lambda o. \lambda s. s \diamond \text{"ignored"} \diamond o$
fullstop	$\mapsto$	$\lambda x. x \diamond \text{"."}$
$\star$	$\mapsto$	function application

### Computing the surface form

$$\begin{aligned}
 & \mathcal{I}_{\text{syn}}(t_{\text{woman}}) \\
 = & (\lambda x. x \diamond \text{"."})((\lambda o. \lambda s. s \diamond \text{"ignored"} \diamond o) \\
 & ((\lambda x. \text{"a"} \diamond x) \text{"woman"}) \text{"John"}) \\
 \hookrightarrow & \text{"John"} \diamond \text{"ignored"} \diamond \text{"a"} \diamond \text{"woman"} \diamond \text{"."}
 \end{aligned}$$

When we *normalize* that term we get what looks like a string, the surface form of our sentence. In ACG tutorials that I read, the fact that we have to normalize, or reduce, the result of the lexicon substitution is hardly ever mentioned. There is a good reason: there is little to say: The calculus here is simply-typed lambda calculus and is strongly normalizing. Every term has the normal form; the normalization is as uneventful as it could ever get. But that would change, in our extension to ACG.

# Haskell demo notable points

## Abstract.hs

- ▶ Defining and *re-using* phrases
- ▶ Type inference
- ▶ The type shows if a phrase is a complete sentence
- ▶ Inferred type shows all features in use
- ▶ Only complete sentence are to be interpreted

We now show how the Greek development we've seen looks in Haskell.

# Outline

Overview

Details

► **Demonstration of the syntax-semantics calculator**

# Applicative

## Function application with ‘side-effects’

- ▶  $i\alpha$  represents a computation that produces the value of the type  $\alpha$  and may have an effect
- ▶ Introduction rule  
 $\text{pure} : \alpha \rightarrow i\alpha$
- ▶ Composing principle
  - ▶ Monad:  $m\alpha \rightarrow (\alpha \rightarrow m\beta) \rightarrow m\beta$
  - ▶ Applicative:  $i(\alpha \rightarrow \beta) \rightarrow i\alpha \rightarrow i\beta$

$\mathcal{I}_{\text{sem}}$  uses effects, and effects are structured through Applicative. You might have heard of monads, an obscure philosophical concept borrowed as a joke into Category theory and rising to prominence through hardly countable monad tutorials. Applicative is a simpler version of monads. For one, ‘applicative’ has been in English language longer (for about 30 years, according to OED: OED quotes 1607 for applicative).

# Applicative

## Function application with ‘side-effects’

- ▶  $i\alpha$  represents a computation that produces the value of the type  $\alpha$  and may have an effect
- ▶ Introduction rule  
 $\text{pure} : \alpha \rightarrow i\alpha$
- ▶ Composing principle
  - ▶ Monad:  $m\alpha \rightarrow (\alpha \rightarrow m\beta) \rightarrow m\beta$
  - ▶ Applicative:  $i(\alpha \rightarrow \beta) \rightarrow i\alpha \rightarrow i\beta$
  
- ▶ All monads are applicatives, but not vice versa
- ▶ Applicatives compose, monads generally not

We'll see many examples of applicatives and their compositions

## Haskell demo

- ▶ **Abstract.hs**: the Abstract language
  - ▶ Defining and *re-using* phrases
  - ▶ Type inference
  - ▶ The type shows if a phrase is a complete sentence
  - ▶ Inferred type shows all features in use
  - ▶ Only complete sentence are to be interpreted
- ▶ **Logic.hs**: the language of Logic; lifting through applicative
- ▶ **Sem.hs**: main transformation; adding ACnj, DynLogic and two levels of quantification – modularly

# Conclusions

## Illustration and extensions of ACG

- ▶ Abstract  $\mapsto$  Syntax & semantics, compositionally
- ▶ Transformations may be composed from smaller ones
- ▶ *Transformation are effectful and non-trivial*

## Mechanical implementation: semantics calculator

- ▶ Implementation in Haskell
- ▶ Applicatives to express effects
- ▶ Towers of applicatives

## Applications

- ▶ Account of dynamic logic
- ▶ Interaction of quantification and pronouns

We have described a computational ACG, emphasizing evaluation as a process to produce a (semantic) logical formula. Computational ACG gives us a principled way to assign different quantifiers different scope-taking abilities, maintaining consistency with Minimalism and avoiding free-wheeling Quantifier Raising.

Computational ACG let us relate quantification and binding: the same mechanism controls the scope of both.

We have implemented Computational ACG by embedding them in Haskell. We can compute ACG yields and, more importantly, denotations. We can do that interactively, in GHCi). There is no longer any need to computing denotations by hand. We (computer, actually) can thus handle more complex examples.