Applicative Abstract Categorial Grammar

Semantics Seminar

Toukyou, Japan, June 1, 2012

Abstract Categorial Grammar (ACG) is a grammar formalism based on typed lambda-calculus. The syntactic side of ACGs – parsing algorithms, generative power and other formal-language–theoretical properties – is under active investigation. ACGs can also elegantly model syntax-semantics interface and compute truth conditions. This semantic side of ACGs has received much less attention.

We describe a generalization of ACGs that lets us give the standard dynamic-logic account of anaphora and analyze quantifier strength, quantifier ambiguity and scope islands. Most of these ACG analyses have not been possible before; prior ACG analyzes of quantifier ambiguity required type lifting, hence higher-order ACGs with very high parsing complexity.

Our generalization to ACG affects only the mapping from abstract language to semantics. We retain all ACG benefits of parsing from the surface form. By avoiding type lifting we keep the order of the abstract signature low, so that parsing remains tractable.

The generalization relies on 'applicative functors', which extend function applications. The fact that applicative functors compose lets us take the full advantage of modularity and compositionality of ACGs. We assemble the semantic mapping from separate, independently developed components responsible for a single phenomenon such as anaphora, coordination, universal or indefinite operator. We have implemented the generalized ACGs in a 'semantic calculator', which is the ordinary Haskell interpreter. The calculator lets us write grammar derivations in a linguist-readable form and see their yields, types and truth conditions. We easily extend fragments with more lexical items and operators, and experiment with different semantic-mapping assemblies.

Phenomena

(1) John ignored Mary and she left.

(2) John ignored a woman and she left.

(3) John ignored every woman and she left.

(3) Every woman ignored a man.

(3) A woman ignored every man.

...

- binding

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

Phenomena

(1) John ignored Mary and she left.

(2) John ignored a woman and she left.

(3) John ignored every woman and she left.

(3) Every woman ignored a man.

(3) A woman ignored every man.

...

- binding
- quantification and binding

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

Phenomena

(1) John ignored Mary and she left.

(2) John ignored a woman and she left.

(3) John ignored every woman and she left.

(3) Every woman ignored a man.

(3) A woman ignored every man.

...

- 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). The last two examples are to illustrate linear and inverse scope and hence quantifier ambiguity.

Phenomena

(1) John ignored Mary and she left.

(2) John ignored a woman and she left.

(3) John ignored every woman and she left.

(3) Every woman ignored a man.

(3) A woman ignored every man.

...

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

ACG as a Semantic *Formalism*

I will use these examples to illustrate the thesis: ACG is a semantic formalism. I stress formalism: just as a grammar formalism lets us conveniently write grammars and express various constraints to reproduce observed natural language phenomena (linguistic competence), semantic formalism lets us conveniently encode constraints and transformations to express the meaning people attach to sentences.

Although some people in the audience are *very* familiar with ACG, to say the least, a bit of overview still seems in order, to introduce the terminology and examples. We start with two intuitions that underlie ACG.

ACG vs CCG

CFG: Categories as Types

John	$: NP$
woman	$: CN$
a	$: CN \rightsquigarrow NP$
ignored	$: NP \rightsquigarrow NP \rightsquigarrow S$

The Merge rule (type schema)

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

$$t : S \quad \equiv \quad t \text{ is accepted}$$

The first intuition comes from Combinatorial Categorial Grammars (CCG). The intuition is interpreting syntactic categories as types and grammatical derivations like type derivations. Suppose we have primitive types NP, CN, S, a few terminals and the following assignment of types to the terminals. This assignment looks a lot like the notation for a Context Free Grammar.

ACG vs CCG

CFG: Categories as Types

John	$: NP$
woman	$: CN$
a	$: CN \rightsquigarrow NP$
ignored	$: NP \rightsquigarrow NP \rightsquigarrow S$

The Merge rule (type schema)

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

$$t : S \quad \equiv \quad t \text{ is accepted}$$

The Merge rule is interpreted as a type schema, essentially the elimination rule for this funny arrow, which is just a binary type constructor. The corresponding $\star$ is an infix composition operator for terms. In most CFG notations, this operator is written as a blank.

ACG vs CCG

CFG: Categories as Types

John	$: NP$
woman	$: CN$
a	$: CN \rightsquigarrow NP$
ignored	$: NP \rightsquigarrow NP \rightsquigarrow S$

The Merge rule (type schema)

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

$$t : S \quad \equiv \quad t \text{ is accepted}$$

ignored $\star$ (a $\star$ woman) $\star$ John

$*$John $\star$ ignored $\star$ (a $\star$ woman)

Too rigid

We may identify the set of typeable terms of the type S as the set of sentences that are recognized/generated by the grammar. Among those sentence is the one one the slide. It seems like a sentence we want to generate for English. There are obvious problems: word order, for example. The sentence we wish to get is untypeable, and there doesn't seem to be an easy way to obtain it. The approach is too rigid.

ACG vs CCG, cont.

`"John"`	: string
`"woman"`	: string
`"a"`	: string
`"ignored"`	: string
Composition	
$\diamondsuit$	: string $\to$ string $\to$ string

$$\texttt{"John"} \diamondsuit \texttt{"ignored"} \diamondsuit (\texttt{"a"} \diamondsuit \texttt{"woman"})$$

$$\texttt{"John"} \diamondsuit \texttt{"John"} \diamondsuit \texttt{"John"}$$

Too flexible

Let's relax the typing and simplify everything to strings. One may read the operation $\diamondsuit$ as string concatenation. The desired sentence comes out. Alas, lots of undesirable sentences also come out. This typing is too loose.

Lexicon

John	$: NP$	`"John"`	$:$ string
woman	$: CN$	`"woman"`	$:$ string
a	$: CN \rightsquigarrow NP$	`"a"`	$:$ string
ignored	$: NP \rightsquigarrow NP \rightsquigarrow S$	`"ignored"`	$:$ string
$\star$	$: (a \rightsquigarrow b) \to a \to b$	$\diamondsuit$	$:$ string $\to$ string $\to$ strin

We may wish for a grammar formalism that lets us 'adjust' the yield of the simple CFG to impose a word order. Lexicon is such an adjustment, mapping constants of the rigid CFG to terms of the loose CFG. Me way also view the lexicon as making the loose CFG more rigid: not all typeable terms of the loose CFG are considered accepted sentences, but only those that are in the image of the lexicon mapping.

Lexicon

John	$: NP$	$\texttt{"John"}$
woman	$: CN$	$\texttt{"woman"}$
a	$: CN \rightsquigarrow NP$	$\lambda x.\, \texttt{"a"} \diamondsuit\ x$
ignored	$: NP \rightsquigarrow NP \rightsquigarrow S$	$\lambda o.\, \lambda s.\, s \diamondsuit\ \texttt{"ignored"} \diamondsuit\ o$
$\star$	$: (a \rightsquigarrow b) \to a \to b$	$\lambda f.\, \lambda x.\, f\ x$

Lexicon maps terms to term generators

Intuition

- The abstract vocabulary specifies the abstract syntactic structure (Curry's tectogrammatics)
- The object vocabulary specifies the possible surface forms
- The lexicon specifies how the abstract structures can be realized into surface forms (Curry's phenogrammatics)

stolen from the ACG course at ESSLLI 2009

To summarize some intuitions.

For the sake of reference we give a bit formal definition of ACG, essentially repeating what we already said but with more jargon and squiggles.

Abstract signature

A higher-order signature

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

Signature Σ_{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 $\rightsquigarrow$)	
$\star$	$: (a \rightsquigarrow b) \rightarrow a \rightarrow b$

Language $\stackrel{\text{def}}{=}$ 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 Σ_{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 $\rightsquigarrow$)	
$\star$	$: (a \rightsquigarrow b) \rightarrow a \rightarrow b$

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

Terms over Σ_{abs}

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

A sample term

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

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 Σ_{str}

Atomic type	string
`"John"`	: string
`"woman"`	: string
`"a"`	: string
`"ignored"`	: string
`"."`	: string
Composition	
$\diamondsuit$	: string $\rightarrow$ string $\rightarrow$ string

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

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

$\mathcal{I}_{\text{syn}}$: mapping of constants of Σ_{abs} to terms over Σ_{str}
λ-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	$\mapsto$	$\lambda o.\, \lambda s.\, s \diamondsuit$ `"ignored"` $\diamondsuit\, o$
fullstop	$\mapsto$	$\lambda x.\, x \diamondsuit$ `"."`
$\star$	$\mapsto$	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 Σ_{abs} to terms over Σ_{str}
λ-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	$\mapsto$	$\lambda o.\, \lambda s.\, s \diamondsuit$ `"ignored"` $\diamondsuit\, o$
fullstop	$\mapsto$	$\lambda x.\, x \diamondsuit$ `"."`
$\star$	$\mapsto$	function application

the surface form

$\mathcal{I}_{\text{syn}}(t_{woman})$

$= \quad (\lambda x.\, x \diamondsuit$ `"."`$)((\lambda o.\, \lambda s.\, s \diamondsuit$ `"ignored"` $\diamondsuit\, o)$
$\qquad ((\lambda x.$ `"a"` $\diamondsuit\, x)$`"woman"`$)$ `"John"`$)$

$\hookrightarrow \quad$ `"John"` $\diamondsuit$ `"ignored"` $\diamondsuit$ `"a"` $\diamondsuit$ `"woman"` $\diamondsuit$ `"."`

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.

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

$\mathcal{I}_{\text{syn}}$: mapping of constants of Σ_{abs} to terms over Σ_{str}
λ-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	$\mapsto$	$\lambda o.\, \lambda s.\, s \diamondsuit$ `"ignored"` $\diamondsuit\, o$
fullstop	$\mapsto$	$\lambda x.\, x \diamondsuit$ `"."`
$\star$	$\mapsto$	function application

Computing the surface form

$\mathcal{I}_{\text{syn}}(t_{woman})$

$= \quad (\lambda x.\, x \diamondsuit$ `"."`$)((\lambda o.\, \lambda s.\, s \diamondsuit$ `"ignored"` $\diamondsuit\, o)$
$\qquad ((\lambda x.$ `"a"` $\diamondsuit\, x)$`"woman"`$)$ `"John"`$)$

$\hookrightarrow \quad$ `"John"` $\diamondsuit$ `"ignored"` $\diamondsuit$ `"a"` $\diamondsuit$ `"woman"` $\diamondsuit$ `"."`

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 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.

Notable points so far

- Compositionality: homomorphic mapping
- Compositionality: several transformations one after another
- Interpretations and interpreting
- Interpret not only constants but also applications; not only base types but also arrows.

We haven't seen much of doing several transformations one after another. We will.

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. I beg the audience to indulge me for a just a bit longer. That talk is about semantics but so far I've been talking about syntax. I'd like to use the simplicity of the examples so far to introduce the 'semantic calculator', which is very handy, including for this talk as we shall see soon.

Now, why we talked so much about syntax at the semantics seminar.
Because meaning (truth conditions) is just another yield of the grammar.
Everything I told you before about syntax applies directly to semantics.

Syntax-semantics interface

Syntax $\longleftrightarrow$ Semantics

Goal: investigate the correspondence between the syntax and semantics

Syntax-semantics interface

Syntax	$\longleftrightarrow$	Semantics
`"John ignored a woman"`		$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

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)

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

Syntax	Semantics
`"John ignored a woman"`	$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

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

Syntax-semantics interface

Abstract

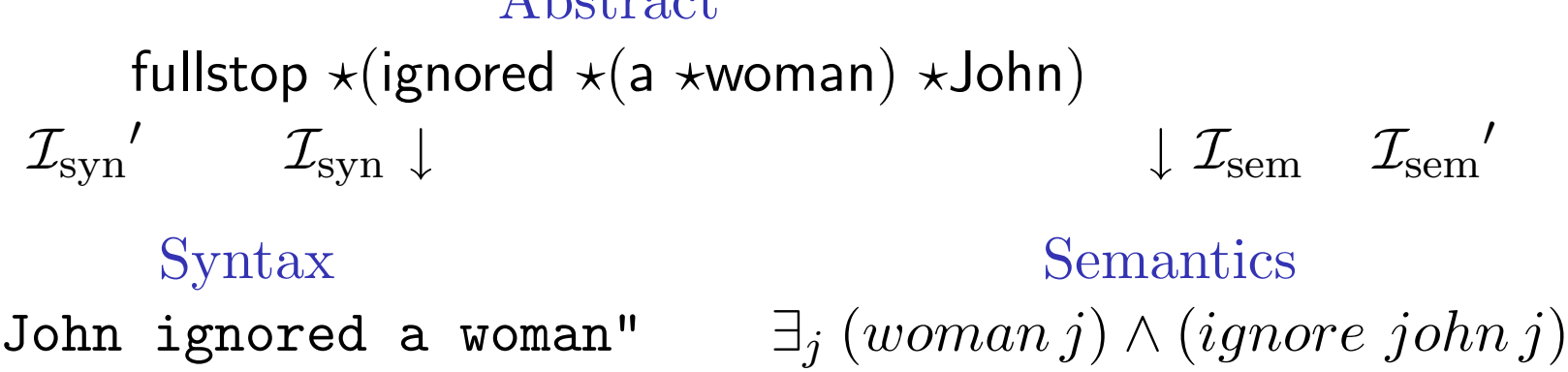
fullstop ⋆(ignored ⋆(a ⋆woman) ⋆John)

Syntax	Semantics
`"John ignored a woman"`	$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

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



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

Syntax-semantics interface

Abstract

fullstop ⋆(ignored ⋆(a ⋆woman) ⋆John)

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

Syntax	Semantics
`"John ignored a woman"`	$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

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 λ-calculus.

Syntax-semantics interface

Abstract

fullstop ⋆(ignored ⋆(a ⋆woman) ⋆John)

$\mathcal{I}_{\text{syn}}$ ↑ ↓ $\mathcal{I}_{\text{sem}}$

Syntax	Semantics
`"John ignored a woman"`	$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

Parsing

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

Syntax-semantics interface

Abstract

fullstop ⋆(ignored ⋆(a ⋆woman) ⋆John)

$\mathcal{I}_{\text{syn}} \downarrow$ $\qquad\qquad$ $\downarrow \not{\mathcal{I}}_{\text{sem}}$

Syntax	Semantics
`"John ignored a woman"`	$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

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

Abstract

fullstop ⋆(ignored ⋆(a ⋆woman) ⋆John)

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

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

$\mathcal{I}$ 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)

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

$\mathcal{I}_{\text{syn}1} \downarrow$ $\quad$ $\downarrow \mathcal{I}_{\text{sem}1}$

Syntax	Semantics
`"John ignored a woman"`	$\exists_j\,(woman\,j) \wedge (ignore\;john\,j)$

$\mathcal{I}$ 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.

Logic language

Signature Σ_{sem}

Atomic types	E, T
$john$	$: E$
$mary$	$: E$
$woman$	$: (ET)$
$ignore$	$: ((EE)T)$
i	$: E$
j	$: E$
$\wedge$	$: ((TT)T)$
$\exists_j$	$: ((ET)T)$
Composition (elimination rule for $(,)$)	
$\circledcirc$	$: (ab) \rightarrow a \rightarrow b$

Let's take an example of mapping of Abstract to semantics. First we need a language in which to express meaning, that is, in which to write truth conditions. The language is a simple predicate logic. The specification of the language looks pretty much like the specification of Abstract, for example: terms, types, the composition rule.

Logic lexicon

$$
\begin{array}{lcl}
\text{NP} & \mapsto & E \\
\text{S, M} & \mapsto & T \\
\text{CN} & \mapsto & (ET) \\
\rightsquigarrow & \mapsto & (,) \\
\text{John} & \mapsto & john \\
\text{woman} & \mapsto & woman \\
\text{ignored} & \mapsto & ignore \\
\star & \mapsto & \circledcirc
\end{array}
$$

$$
\begin{array}{ll}
 & (\text{ignored} \star \text{ Mary}) \star \text{ John} \\
\hookrightarrow & (ignore \circledcirc\ mary) \circledcirc\ john
\end{array}
$$

Here is the mapping from the abstract syntax to semantics. It is most trivial so far, a mere replacement of constants and stars (the composition operator), How to deal with pronouns and quantification, however? We now introduce the tool that permits more interesting transformations, beyond mere substitution.

Applicative

Function application with 'side-effects'

- $i\alpha$ represents a computation that produces the value of the type α and may have en affect
- Introduction rule
  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 (by 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 α and may have en affect
- Introduction rule
  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

Applicative logic lexicon

NP	$\mapsto$	iE
S, M	$\mapsto$	iT
CN	$\mapsto$	$i(ET)$
$\rightsquigarrow$	$\mapsto$	$i(,)$
John	$\mapsto$	pure $john$
woman	$\mapsto$	pure $woman$
ignored	$\mapsto$	pure $ignore$
x $\star$ y	$\mapsto$	(pure$\odot$) $\circledast$ x $\circledast$ y

i is intentionally left unspecified

We re-write our mapping from the abstract syntax to semantics to operate in an applicative – *any* applicative. The mapping on this slide remains valid no matter which applicative we use, or how many.

Null applicative: Logic

$$i_0\ \alpha = \alpha$$

$$\text{pure} : \alpha \to i_0\alpha$$

$$\text{pure} = \lambda x.\, x$$

$$(\circledast) : i_0(\alpha \to \beta) \to i_0\alpha \to i_0\beta$$

$$e_1 \circledast e_2 = e_1 e_2$$

(ignored $\star$ Mary) $\star$ John

$\hookrightarrow$ $(ignore \circledcirc\ mary) \circledcirc\ john$

The first applicative does nothing at all: the applicative application is the ordinary function application. Using this applicative with the lexicon we have just seen reproduces the old result. Now we can generalize however. Let's add pronouns, and a state applicative.

State applicative: Pronouns

$$
\begin{aligned}
i_S\ \alpha &= \sigma \to \alpha * \sigma \\
\text{pure} &: \alpha \to i_S \alpha \\
\text{pure} &= \lambda x.\, \lambda s.(x, s) \\
(\circledast) &: i_S(\alpha \to \beta) \to i_S \alpha \to i_S \beta \\
e_1 \circledast e_2 &= \lambda s.(\lambda(v_1, s_1). \\
&\qquad (\lambda(v_2, s_2).(v_1 v_2, s_2))(e_2\ s_1))(e_1\ s)
\end{aligned}
$$

Here is the state applicative with the state σ. For pronouns, we choose σ to be the stack of available antecedents, from which a pronoun selects. If the stack is empty, the mapping fails. The mapping is inherently partial.

State applicative: Pronouns

$$i_S\ \alpha = \sigma \to \alpha * \sigma$$

$$\text{pure} : \alpha \to i_S\alpha$$

$$\text{pure} = \lambda x.\, \lambda s.(x, s)$$

$$(\circledast) : i_S(\alpha \to \beta) \to i_S\alpha \to i_S\beta$$

$$e_1 \circledast e_2 = \lambda s.(\lambda(v_1, s_1).$$
$$(\lambda(v_2, s_2).(v_1 v_2, s_2))(e_2\ s_1))(e_1\ s)$$

$$\text{fullstop} \star\ (\text{and} \star (\text{ignored} \star \text{Mary} \star \text{John}) \star (\text{left} \star \text{she}))$$

$$\hookrightarrow\quad ((ignore \circledcirc\ mary) \circledcirc\ john) \wedge (leave \circledcirc mary)$$

$$\text{left} \star \text{she}$$

$$\not\hookrightarrow$$

It becomes too tedious to do the reductions by hand. Let's use our semantic calculator. See `Sem.hs`. One may view this applicative as re-writing the language with pronouns into the language without them.

CPS applicative: Quantification

`Sem.hs`

We re-write the language with quantifiers into the language without quantifiers. The target language may still have pronouns, which are taken care at the next step, as we have already discussed.

Two CPS applicatives: direct and inverse scope

`Sem.hs`

We don't have to limit ourselves with one level of quantification. We can add two levels, one for universals and one for indefinites (existentials). We take care of existentials, and then take care of universals, and then of pronouns. The entire semantic lexicon is composed from small transformation steps. The assignment of quantifiers to different layers lets us analyze both linear and inverse scope. See `Sem.hs` for details.

Two CPS and the State applicatives

`Sem.hs`

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

ACG: a grammar formalism, a *semantic formalism*

- Abstract $\mapsto$ Syntax & semantics, compositionally
- Transformations may be composed from smaller ones
- *Transformation are effectful and non-trivial*
- *Interpret not only constants but also applications; not only base types but also arrows.*

Mechanical implementation: semantics calculator

- Applicatives to express effects
- Towers of applicatives

Applications

- Account of dynamic logic
- Interaction of quantification and pronouns

`http://okmij.org/ftp/gengo/applicative-symantics/`

ACG not only the grammar formalism, it is also a semantic formalism
We have described an applicative ACG, emphasizing evaluation as a process to produce a (semantic) logical formula. Applicative ACG gives us a principled way to assign different quantifiers different scope-taking abilities, maintaining consistency with Minimalism and avoiding free-wheeling Quantifier Raising.
Applicative ACG let us relate quantification and binding: the same mechanism controls the scope of both.

We have implemented the Applicative ACGs 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.