A glue semantics parser and prover

Mark-Matthias Zymla
Moritz Messmer

15/12/2017

1. About

2. Introduction to glue semantics

3. Hepple-style chart prover
    - first-order prover
    - higher-order prover

4. Generating lexical entries

5. Conclusion

Outline

1. About
2. Introduction to glue semantics
3. Hepple-style chart prover
   - first-order prover
   - higher-order prover
4. Generating lexical entries
5. Conclusion

About

- 4-week mini project with experts in California (Dick Crouch, Tracy Holloway King)
- **GOAL:** Implementing a semantic parser based on glue semantics in Java
- Some existing resources:
  - NLTK computational semantics package (written in Python)
  - Glue implementation PARC by Richard Crouch and colleagues (written in Prolog)

  → Served as initial guiding points

Outline

1. About
2. Introduction to glue semantics
3. Hepple-style chart prover
   - first-order prover
   - higher-order prover
4. Generating lexical entries
5. Conclusion

> “[glue semantics] is an approach to the semantic interpretation of natural language that uses a fragment of linear logic as a deductive glue for combining together the meanings of words and phrases”
> – Crouch & van Genabith (2000)

- Lexical entries consist of two elements:
  - **Glue language:** Linear logic – can be understood as semantic types (Curry-Howard-isomorphism)
  - **Meaning language** Montague style semantics (but other formalism are possible)
- ex.: $\lambda x.sleep(x) : A \multimap B$

The appeal of linear logic

- Linear logic (LL) is a *resource-conscious* logic
  premises, assumptions and conclusions as used in logical proofs are resources (not truths or facts)

$$A, A \rightarrow B, A \rightarrow C \models A, C \text{ vs.} A, A \multimap B, A \rightarrow C \not\models A, C$$

- The syntax of proof systems is not always in one-to-one correspondence to the underlying proof object

→ LL better suited to describe underlying proof objects

- Resource usage occurs in natural language: Words and phrases correspond to resources
- A sentence denotes a successful linear logic proof

Relevant rules

- We use the *implicational framgment* of linear logic

Introduction rule

$$\dfrac{\begin{array}{c}[x : A]^i\\ \vdots\\ f(x) : B\end{array}}{\lambda x.f(x) : A \multimap B}\multimap_{I,i}$$

Elimination rule

$$\dfrac{f : A \multimap B \qquad a : A}{f(a) : B}\multimap_{E}$$

Semantic composition as proof

- *John loves Mary.*
- Lexical entries:
    - $[\![\text{John}]\!] = j : g$
    - $[\![\text{Mary}]\!] = m : h$
    - $[\![\text{loves}]\!] = \lambda x.\lambda y.loves(x, y) : g \multimap (h \multimap f)$

$$\dfrac{\dfrac{\lambda x.\lambda y.loves(x, y) : g \multimap (h \multimap f) \qquad j : g}{\lambda y.loves(j, y) : gh \multimap f} \qquad m : h}{loves(j, m) : f}$$

From syntax to semantics

$$\begin{bmatrix} \text{PRED} & \text{'love<John,Mary>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'John'} \end{bmatrix} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \end{bmatrix} \end{bmatrix}$$

- $\lambda x.\lambda y.loves(x,y) :$ $\uparrow .SUBJ \multimap (\uparrow .OBJ \multimap \uparrow)$
- $j : \uparrow .SUBJ$
- $m : \uparrow .OBJ$

- $\uparrow$ refers to a specific f-structure node (e.g. $\uparrow$ points to the f-structure of the whole sentence; $\uparrow .SUBJ$ points to the f-structure node of the subject)
- Syntactic analysis determines linear logic resources

From syntax to semantics

- *A big dog chases every cat.*

$$\left[\text{SUBJ} \quad \left[\begin{array}{ll} \text{PRED} & \text{'dog'} \\ \text{DET} & \left[\text{PRED 'a'}\right] \\ \text{ADJUNCT} & \left\{\text{PRED 'big'}\right\} \end{array}\right]\right]$$

- $\lambda P.\lambda Q.\exists x[P(x) \wedge Q(x)]$:
  $(g \multimap \uparrow .SUBJ) \multimap ((\uparrow .OBJ \multimap \uparrow) \multimap \uparrow)$
- $\lambda x.dog(x) : g \multimap \uparrow .SUBJ$
- $\lambda P.\lambda x.big(x) \wedge P(x) : (g \multimap \uparrow .SUBJ) \multimap (g \multimap \uparrow .SUBJ)$

→ $\lambda Q.\exists x[(big(x) \wedge dog(x)) \wedge Q(x)] : ((\uparrow .OBJ \multimap \uparrow) \multimap \uparrow)$

From syntax to semantics

- *A big dog chases every cat.*

$$\begin{bmatrix} \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'cat'} \\ \text{DET} & \begin{bmatrix} \text{PRED} & \text{'every'} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

- $\lambda P.\lambda Q.\forall y[P(y) \rightarrow Q(y)]$: $(h \multimap \uparrow .OBJ) \multimap ((\uparrow .SUBJ \multimap \uparrow) \multimap \uparrow)$
- $\lambda x.cat(x)$: $h \multimap \uparrow .SUBJ$

$\rightarrow$ $\lambda Q.\forall y[cat(y) \rightarrow Q(y)]$: $((\uparrow .SUBJ \multimap \uparrow) \multimap \uparrow)$

- **What happened?**
  - Quantifiers have the template: $(x \multimap RESTR) \multimap ((SCOPE \multimap \uparrow) \multimap \uparrow)$.
  - The RESTR and SCOPE of a quantifier a determined by the Syntax.

- $[\![\text{a big dog}]\!] = \lambda Q.\exists x[(big(x) \wedge dog(x)) \wedge Q(x)] : ((h \multimap f) \multimap f)$
- $[\![\text{every cat}]\!] = \lambda Q.\forall y[cat(y) \rightarrow Q(y)] : ((g \multimap f) \multimap f)$
- $[\![\text{chases}]\!] = \lambda x.\lambda y.chases(x, y) : h \multimap (g \multimap f)$

$$\dfrac{\dfrac{\dfrac{\dfrac{[h]^1 \quad h \multimap (g \multimap f)}{g \multimap f}\,\multimap_E \quad (g \multimap f) \multimap f}{f}\,\multimap_E}{h \multimap f}\,\multimap_{I,1} \quad (h \multimap (f \multimap f)}{f}\,\multimap_E$$

- $\lambda x.\lambda y.chases(x, y) : h \multimap (g \multimap f)\ (z) = \lambda y.chases(z, y)$
- $[\![\text{every cat}]\!]\ (\lambda y.chases(z, y)) = \forall x[cat(x) \rightarrow chases(z, y)]$
- $\forall x[cat(x) \rightarrow chases(z, x)]$ $=_{\multimap_{I,i}} \lambda z.\forall x[cat(x) \rightarrow chases(z, x)]$
- $[\![\text{a big dog}]\!]\ ([\![\text{ every cat}]\!]) =$ $\exists y[(big(y) \wedge dog(y)) \rightarrow \forall x[cat(x) \wedge chases(y, x)]]$

- $[\![\text{a big dog}]\!] = \lambda Q.\exists x[(big(x) \wedge dog(x)) \wedge Q(x)] : ((h \multimap f) \multimap f)$
- $[\![\text{every cat}]\!] = \lambda Q.\forall y[dog(y) \rightarrow Q(y)] : ((g \multimap f) \multimap f)$
- $[\![\text{chases}]\!] = \lambda x.\lambda y.chases(x, y) : h \multimap (g \multimap f)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[g]^2 \quad \dfrac{[h]^1 \quad h \multimap (g \multimap f)}{g \multimap f}\multimap_E}{f}\multimap_E}{h \multimap f}\multimap_{I,1} \quad (h \multimap f) \multimap f}{f}\multimap_E}{g \multimap f}\multimap_{I,2} \quad (g \multimap f) \multimap h}{f}\multimap_E$$

- Homework: Prove that this works on the lambda-side!

Outline

1. About
2. Introduction to glue semantics
3. Hepple-style chart prover
    - first-order prover
    - higher-order prover
4. Generating lexical entries
5. Conclusion

Hepple (1996) Chart prover

- Chart parsers store partial results and re-use them to prevent backtracking
- Hepple's system uses same idea
- First step: first-order chart parser without hypothetical reasoning (no $\multimap$-introduction and no assumptions)
  - linear use of resources enforced by using indexes
  - each premise assigned unique index
  - when combining premises their index sets are unified
  - two premises can only be combined when their index sets are disjoint

Outline

1 About

2 Introduction to glue semantics

3 Hepple-style chart prover
- first-order prover
- higher-order prover

4 Generating lexical entries

5 Conclusion

first-order chart prover pseudo code

```
Stack A (agenda)
List D (database)
for A contains premises do
    pop premise $P_A$
    add $P_A$ to D
    for all Premises $P_D$ in D do
        if $P_A$ and $P_D$ combineable and index sets disjoint then
            add new combined premise to A
        end if
    end for
end for
if any $P_D$ from D has a full set of indexes it is a valid solution
```

Outline

1. About
2. Introduction to glue semantics
3. Hepple-style chart prover
   - first-order prover
   - higher-order prover
4. Generating lexical entries
5. Conclusion

higher-order chart prover

- algorithm so far only works for formulas of the form $A_a \multimap B_c$, where a is an atom
- higher-order formulas with nested consumers usually require $\multimap$-introduction
- hypothetical reasoning makes computation very complex
- Hepple's solution: transform the initial (potentially higher-order) formulas into a set of first-order formulas
- nested consumers are "compiled out" to additional assumptions:
  $(a \multimap b) \multimap c \Rightarrow$

higher-order chart prover

- algorithm so far only works for formulas of the form $\mathrm{A}_a \multimap \mathrm{B}_c$, where a is an atom
- higher-order formulas with nested consumers usually require $\multimap$-introduction
- hypothetical reasoning makes computation very complex
- Hepple's solution: transform the initial (potentially higher-order) formulas into a set of first-order formulas
- nested consumers are "compiled out" to additional assumptions:
  $(a \multimap b) \multimap c \Rightarrow b[a] \multimap c;\ \{a\}$

higher-order chart prover

- extracted assumptions are marked as such (notated with {}) and assigned a new unique index
- formula from which assumption is extracted gets extracted resource as discharge (notated with [])
- when two premises are combined the following rules apply:

higher-order chart prover

- extracted assumptions are marked as such (notated with {}) and assigned a new unique index
- formula from which assumption is extracted gets extracted resource as discharge (notated with [])
- when two premises are combined the following rules apply:
  - if one or both premises contain assumptions, these are added to the set of assumptions of the combined premise
  - if a premise contains discharges, the set of assumptions of the other premise must contain the dischcarged resource
  - matched assumption and discharge pairs are removed from the book-keeping
- on the meaning side, a compilation step amounts to functional application with a deliberate "accidental binding" of the relevant variable

compilation and combination of higher-order formula

(1) Everybody sleeps.

original premises:

$g_1 \multimap f \quad : \lambda \mathrm{y.sleep(y)}$

$(g_2 \multimap H) \multimap H \quad : \lambda \mathrm{P}.\forall \mathrm{x}[\mathrm{person(x)} \wedge \mathrm{P(x)}]$

compiled premises:

$g_1 \multimap f \quad : \lambda \mathrm{y.sleep(y)}$

$\{g_2\} \quad : \mathrm{v}$

$H[g_2] \multimap H \quad : \lambda \mathrm{u}.\lambda \mathrm{P}.\forall \mathrm{x}[\mathrm{person(x)} \wedge \mathrm{P(x)}](\lambda \mathrm{v.u})$

$$\dfrac{\dfrac{H[g_2] \multimap H : \lambda \mathrm{u}.\lambda \mathrm{P}.\forall \mathrm{x}[\mathrm{person(x)} \wedge \mathrm{P(x)}](\lambda \mathrm{v.u}) \qquad \dfrac{g_1 \multimap f : \lambda \mathrm{y.sleep(y)} \quad \{g_2\} : \mathrm{v}}{f\{g_2\} : \mathrm{sleep(v)}}}{f : \lambda \mathrm{P}.\forall \mathrm{x}[\mathrm{person(x)} \wedge \mathrm{P(x)}](\lambda \mathrm{v.sleep(v)})}\,[\mathrm{H/f}]}{f : \forall \mathrm{x}[\mathrm{person(x)} \wedge \mathrm{sleep(x)}]}\,\beta\text{-conversion}$$

pseudo code: higher-order prover

```
Stack A (agenda)
List D (database)
Solutions S (all premises with full index sets)
for A contains premises do
    pop premise $P_A$
    add $P_A$ to D
    for all Premises $P_D$ in D do
        if $P_A$ and $P_D$ combineable and index sets disjoint then
            if $P_A$ and/or $P_D$ contain assumptions then
                combine sets of assumptions
                add new combined premise to A
            else if $P_A$ or $P_D$ contain discharges then
                if discharges are a subset of assumptions then
                    delete "used" discharges and assumptions
                    add new combined premise to A
                end if
            else
                no assumptions or discharges; combine premises as usual
            end if
        end if
    end for
end for
```

special case: transforming premises

- terms of the form $A_a \multimap B_c$ don't need compilation, only as long as B is not left-nested
- terms like (2) need to be compiled, even though the algorithm so far would treat them as first-order
- resources may be swapped to derive the equivalent term in (3)

(2) $i \multimap ((g \multimap H) \multimap H)$

(3) $(g \multimap H) \multimap (i \multimap H)$

(4) $\{g\}$
$H[g] \multimap (i \multimap H)$

special case: transforming premises

On the semantic side this amounts to swapping the two outermost lambdas

(5) $i \multimap ((g \multimap H) \multimap H) : \lambda P.\lambda Q.\forall x[P(x) \wedge Q(x)]$

(6) $(g \multimap H) \multimap (i \multimap H) : \lambda Q.\lambda P.\forall x[P(x) \wedge Q(x)]$

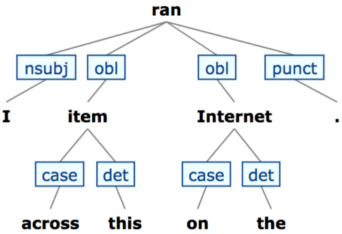
(7) $\{g\} : v$
$H[g] \multimap (i \multimap H) : \lambda u.\lambda Q.\lambda P.\forall x[P(x) \wedge Q(x)](\lambda v.u)$

Outline

1. About
2. Introduction to glue semantics
3. Hepple-style chart prover
   - first-order prover
   - higher-order prover
4. Generating lexical entries
5. Conclusion

From dependencies to glue premises

- Other than LFG structures, dependency parsers are not inherently flat.



- In LFG we made use of the flat f-structure to determine relations between syntax and semantics

→ We need to flatten out the dependency structure.

from dependencies to glue premises

- As a flat structure we use a hashmap with indices as keys.
  - (0 ran)
  - (0 nsubj I)
  - (0 obl 1)
  - (1 item)
  - (1 case across)
  - (1 det this)
  - (0 obl 2)
  - (2 Internet)
  - (2 case on)
  - (2 det the)
- The same process can be conducted on an f-structure.
- certain dependencies directly receive a lexical entry, e.g.
  - $nsubj(\%\%) \wedge nn(\%\%) \rightarrow g_{subj} : \lambda x.\%\%(x)$
  - if (0 %%) has nsubj(%) $\rightarrow \lambda x.\%\%(x)$
  - if (0 %%) has nsubj(%) and nobj(%')$\rightarrow \lambda x.\lambda y.\%\%(x, y)$

from dependencies to glue premises

Determiners

- The template for quantifiers is: $(x \multimap RESTR) \multimap ((SCOPE \multimap \uparrow) \multimap \uparrow)$.
- The restrictor is always the dependency that governs the quantifier
- The scope is newly instantiated for a quantifier and later unified with the arguments of the verb.
  - g: $(x \multimap SUBJ) \multimap ((SCOPE_A \multimap \uparrow) \multimap \uparrow)$
  - h: $(x \multimap OBJ) \multimap ((SCOPE_B \multimap \uparrow) \multimap \uparrow)$
  - $g \multimap (h \multimap f)$: $SCOPE_A \multimap (SCOPE_B \multimap \uparrow)$

Outline

1. About
2. Introduction to glue semantics
3. Hepple-style chart prover
    - first-order prover
    - higher-order prover
4. Generating lexical entries
5. Conclusion

Conclusion

- We presented a semantic parser at the core of which is a chart parser for linear logic formulas that decomposes higher order linear logic formulas into first order formulas
- We implemented corresponding semantics that can be applied to natural language
- We implemented a small system for translating dependency parses into semantic premises that can be proven/composed with the parser
- Time for a **DEMO**