FUNDAÇÃO GETULIO VARGAS

ESCOLA DE MATEMÁTICA APLICADA - FGV/EMAp

CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA

Semântica Computacional para Textos Normativos

por

Guilherme Paulino Passos

Rio de Janeiro

2016

FUNDAÇÃO GETULIO VARGAS

ESCOLA DE MATEMÁTICA APLICADA - FGV/EMAp

CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA

Semântica Computacional para Textos Normativos

”Declaro ser o único autor do presente projeto de monografia que refere-se ao plano de trabalho a ser executado para continuidade da monografia e ressalto que não recorri a qualquer forma de colaboração ou auxílio de terceiros para realizá-lo a não ser nos casos e para os fins autorizados pelo professor orientador”

Guilherme Paulino Passos

Orientador: Prof. Dr. Alexandre Rademaker

Rio de Janeiro

2015

GUILHERME PAULINO PASSOS

Semântica Computacional para Textos Normativos

"Monografia apresentada à Escola de Matemática Aplicada - FGV/EMAp como requisito parcial para a obtenção do grau de Bacharel em Matemática Aplicada."

Aprovado em ____ de _____ de _____ .

Grau atribuido à Monografia: ____ .

Professor Orientador: Prof. Dr. Alexandre Rademaker

Escola de Matemática Aplicada

Fundação Getulio Vargas

Professor Tutor: Prof. Dr. Paulo Cezar Pinto Carvalho

Escola de Matemática Aplicada

Fundação Getulio Vargas

Contents

1 Introdução

Não esquecer
le dizer que
linguagem-
bjeto natural
erá o inglês

2 Representação semântica

2.1 Introdução

Desejamos associar a cada expressão de linguagem natural um significado formal, simbólico. Além disso, desejamos fazê-lo de modo algorítmico, que possa ser reproduzido por um computador.

A linguagem formal que utilizaremos para representar o significado de frases é *lógica de primeira ordem*. Jurafsky and Martin (2009) apresentam como propriedades interessantes para representações: verificabilidade, representações não ambíguas, existência de uma forma canônica, capacidade de inferência e uso de variáveis e expressividade. Todas estas são possuídas pela lógica de primeira ordem. Também uma interessante propriedade da lógica de primeira ordem é sua relativa intuitividade. Bastando explicar o que significam os símbolos conectivos (como $\wedge$ (significando "*e*") e $\rightarrow$ (significando "*se* ... *então* ...")), bem como os quantificadores ($\forall$ ("*para todo*") e $\exists$ ("*existe*"), uma expressão formal em lógica é compreensível .

Ainda que tenhamos escolhido a lógica de primeira ordem para ser a linguagem das representações semânticas para frases, isto não nos informa qual deve ser a representação semântica de palavras e expressões menores. Talvez algumas poderiam ser feitas por termos, mas não está de todo claro qual seria o significado de uma expressão como "*to run*" ("*correr*") ou "*that walks*" ("*que anda*").

Em nossos pressupostos, adotamos o *Princípio da Composicionalidade*. Segundo o mesmo, o significado de expressões complexas é função das expressões mais simples que a compõem. Em um exemplo como "*Caim kills Abel*", isto nos informa que o significado desta frase depende do significado de "*Caim*", "*kills*" e "*Abel*". Entretanto, isto não nos diz como funciona esta dependência, ou a função que leva o significado das expressões simples ao da expressão complexa.

Por exemplo, podemos entender que o significado de "*kills*" é o predicado binário $kill(\ldots,\ldots)$, onde convencionamos que o primeiro argumento é o agressor (isto é, aquele que mata) e o segundo argumento é a vítima (aquele que é morto). Também podemos entender os significados de "*Caim*" e "*Abel*" como as constantes *caim* e *abel*, respectivamente. Assim, apesar de $kill(abel, caim$ ser formada com o significado destes três termos, respeitando a composicionalidade, esta não é a expressão que queremos, e sim

Conferir se falo de lógica aqui ou se puxo isso para a introdução.

é verdade?

$kill(caim, abel)$.

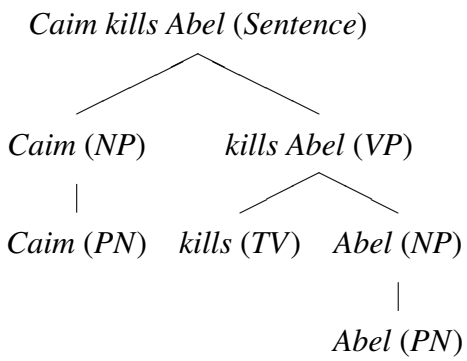
O que nos falta é a *sintaxe*. A sintaxe é o conjunto de regras e processos que organizam a estrutura de frases.

achar uma boa referência

Assim, as palavras em uma frase existem em relação a uma certa estrutura, que é essencial para capturar o significado. No inglês, com a estrutura *Sujeito - Verbo - Predicado*, entendemos que "*Caim kills Abel*" significa $kill(caim, abel)$, e não $kill(abel, caim)$.

O foco deste trabalho não é na sintaxe, de modo que utilizamos uma sintaxe simples: a gramática é implementada pelo mecanismo de Gramática de Cláusulas Definidas (*Definite Clause Grammar* - DCG). A análise sintática é feita na forma de uma árvore cujos nós que são folhas são categorias sintáticas básicas (tais como sujeito (*noun*), verbo transitivo (*transitive verb*) e quantificador (*quantifier*, considerado caso particular de *determiner*). Já os nós que não são folhas representam categorias sintáticas complexas (tais como sintagma nominal (*noun phrase*) ou sintagma verbal (*verb phrase*). (Blackburn and Bos, 2005, p. 58)

Um exemplo de tal árvore, para a frase "*Caim kills Abel*", seria:



Aqui, temos as classes sintáticas:

NP – noun phrase (sintagma nominal)

PN – proper noun (nome próprio)

VP – verb phrase (sintagma verbal)

TV – transitive verb (verbo transitivo)

A decomposição parece lingüisticamente razoável, bem como útil para a compreensão do significado. Resta saber, assim, como podemos elaborar a construção da semântica de uma frase completa a partir de tal análise sintática e dos significados dos termos mais elementares. Essa idéia nos seguirá pelo restante do trabalho, permitindo

separar nossas análises, bem como nossos códigos, pela seguinte idéia: a sintaxe da nossa linguagem natural objeto pode ser separada em léxico, a análise de palavras ou expressões em si, como unidades básicas; e em gramática, a análise de como as classes sintáticas se compõem para formar novas, bem como outras relações de concordância (como concordância de gênero ou de número). Já a semântica também pode ser tratada a nível de léxico, em que cada classe sintática básica terá um modelo próprio de interpretação semântica; bem como a nível de gramática, em que a semântica de uma expressão complexa será formada por uma forma de composição entre as semânticas das expressões que a constituem.

2.2 Cálculo Lambda

Para realizar um método sistemático de composição dos significados, é introduzido o formalismo do *cálculo lambda*. Aqui, ele será uma extensão da linguagem da lógica de primeira ordem. Dois símbolos novos serão introduzidos: o símbolo de abstração "λ" e o de aplicação "@".

O símbolo "λ" será um operador sobre variáveis, permitindo a "captura" das mesmas, do mesmo modo a que um quantificador (como "$\forall$"). Por exemplo, sendo $man(x)$ uma fórmula de primeira ordem, $\lambda x.man(x)$ é uma fórmula do nosso cálculo lambda, em que a variável x está capturada pelo operador λ; alternativamente, $\lambda x.$ está *abstraindo sobre* x.

Por sua vez, o símbolo "@", que conecta duas fórmulas de cálculo lambda, representa uma *aplicação*. Assim, se F e A são duas fórmulas de cálculo lambda, $F@A$ é também uma fórmula de cálculo lambda, chamada uma *aplicação funcional* de F em A, ou uma aplicação na qual F é um *funtor* e A é o *argumento*. Por exemplo, em $\lambda x.man(x)@john$, o funtor é $\lambda x.man(x)$ e o argumento é $john$.

Uma expressão de aplicação funcional representa o comando de aplicar o argumento no funtor, que usualmente será prefixado por uma abstração. A interpretação desse comando é: retire o prefixo de abstração do funtor e, em toda ocorrência da variável abstraída, a substitua pelo argumento da aplicação. Por exemplo, em $\lambda x.man(x)@john$, o funtor é $\lambda x.man(x)$ e a interpretação do comando é de retirar o prefixo $\lambda x.$ e substituir toda ocorrência de x no funtor pelo argumento $john$, o que produz o resultado de $man(john)$. Transformar uma aplicação em sua fórmula resultante após o processo de aplicação é

uma operação chamada de β-*redução*, β-*conversão* ou λ-*conversão*. (Blackburn and Bos, 2005, p. 67)

Destacamos que aplicações podem ser subfórmulas de outras fórmulas, com a β-redução da fórmula maior sendo a β-redução de suas subfórmulas, bem como que não é necessário ser um termo ou uma variável para ser um argumento de uma aplicação. Veja este exemplo: É bem formada a fórmula $(\lambda P.P@mia)@\lambda x.woman(x)$. Em uma primeira etapa de β-redução, chegamos à fórmula $\lambda x.woman(x)@mia$ e aí, mais uma vez realizando a operação, chegamos à sua β-redução final $woman(mia)$.

Um cuidado a se ter é que pode ser necessário trocar o símbolo das variáveis em uma aplicação. É suficiente trocar todas as variáveis ligadas (isto é, capturadas por um operador) do funtor por variáveis novas, não utilizadas até então. A operação de substituir todas as variáveis ligadas por outras é chamada de α-*conversão*, enquanto se uma fórmula pode ser gerada através de α-conversão de outra, as duas fórmulas são ditas α-*equivalentes*. Para um exemplo em que não realizar a α-conversão antes de uma β-conversão pode gerar problemas, basta realizar a β-conversão da seguinte expressão: $\lambda x.\exists y.not\_equal(x,y)@y$. O resultado incorreto seria $\exists y.not\_equal(y,y)$, enquanto o resultado adequado seria $\exists y.not\_equal(z,y)$.

Desse modo, temos o cálculo lambda como uma "linguagem de cola", permitindo fazer composições de expressões até gerar verdadeiras expressões de primeira ordem. A abordagem então é criar, de algum modo, a representação semântica a nível de léxico (isto é, a nível de classes sintáticas básicas), bem como montar a representação semântica a nível da gramática, pela composição de termos mais simples, de algum modo compatível com a semântica a nível lexical. Vejamos alguns exemplos:

Para nomes próprios (*proper names*), a semântica é: $\lambda u.u@symbol$, onde *symbol* representa o símbolo do nome próprio (por exemplo, *john*).

Por sua vez, para verbos transitivos temos a semântica $\lambda k.\lambda y.k@(\lambda x.symbol(y,x))$, onde mais uma vez *symbol* reprenta o símbolo específico da palavra (por exemplo, *kill*).

Pensemos agora no sintagma verbal (*verb phrase*) "*kills Abel*". Um modo natural de pensar na composição é, sendo *A* a expressão semântica de "*kill*" e *B*, a de "*Abel*",

realizar a aplicação $A@B$. Com efeito, fazendo isso teríamos:

$$(\lambda k.\lambda y.k@(\lambda x.kill(y,x)))@\lambda u.u@abel$$
$$\lambda y.((\lambda u.u@abel)@(\lambda x.kill(y,x)))$$
$$\lambda y.(\lambda x.kill(y,x)@abel)$$
$$\lambda y.kill(y,abel)$$

Agora, podemos juntar o sintagma nominal (e também nome próprio) "*Caim*" e o sintagma verbal "*kills Abel*", aplicando a semântica do segundo na do primeiro, de onde teríamos:

$$(\lambda u.u@caim)@(\lambda y.kill(y,abel))$$
$$(\lambda y.kill(y,abel))@caim$$
$$kill(caim,abel)$$

Assim, chegamos a uma representação da frase "*Caim kills Abel*" que é uma expressão de lógica de primeira ordem, utilizando o cálculo lambda como ferramenta para composição sistemática do sentido de expressões menores.

2.3 Armazenamento de Cooper

Apesar deste método produzir resultados interessantes, ele não é suficiente. Uma característica particular é que, do modo que realizamos, cada decomposição sintática está associada a apenas uma possibilidade semântica. Isto não quer dizer que o modelo até então não consegue tratar de ambigüidades.

Em primeiro lugar, as ambigüidades lexicais podem ser tratadas colocando em nosso sistema todos os sentidos possíveis de determinada expressão. Assim, homógrafos (palavras com a mesma grafia mas significados distintos) podem ser considerados como entradas distintas em nosso banco de dados da semântica lexical. Um uso interessante da linguagem Prolog está no fato de que a mesma possibilita a geração de diversos resultados possíveis, pelo mecanismo de *backtracking*. Assim, a implementação em Prolog permite que a semântica a nível léxico seja capturada. Em segundo lugar, ambigüidades por diferentes possibilidades de decomposição sintática de uma mesa frase também podem ser tratadas pelo modelo até então. Novamente, a implementação se beneficia do

mecanismo de *backtracking* do Prolog, de modo que diferentes decomposições sintáticas e seus significados associados podem ser gerados sucessivamente.

Entretanto, podemos apontar um tipo de ambigüidade que, até então, nosso modelo é incapaz de tratar: as ditas *ambigüidades de escopo*. As ambigüidades de escopo são melhor explicadas através de exemplo. Analise a frase: "*Every judge hates an attorney.*"

Esta frase parece ter duas interpretações possíveis: na primeira, para cada juiz existe um advogado odiado por aquele. Possivelmente, são advogados distintos. Já na segunda leitura, existe um advogado que é odiado por todos os juízes: isto é, o mesmo advogado!

Essa dúvida parece ser gerada pelo *escopo* dos quantificadores "*every*" e "*a*". Caso o quantificador "*every*" seja *mais externo* (ou *out-scoping*) ao quantificador "*a*", então teremos a primeira leitura. Neste caso, também dizemos que o quantificador "*every*" tem *escopo sobre* o quantificador "*a*". Por outro lado, caso o quantificador "*a*" tenha escopo sobre o quantificador "*every*", a leitura será a segunda. Perceba que, ao que parece, as ambigüidades de escopo não são geradas por, realmente, análises sintáticas distintas, mas sim por uma dificuldade de atribuição de significado à uma decomposição sintática em particular.

Notemos também que a ocorrência de quantificadores gera seus problemas na função sintática de sintagma nominal (*noun phrase*), pois a combinação quantificador e substantivo (*determiner* + *noun*) ocorre apenas nela. Isso sugere que alteremos o modo pelo qual tratamos a semântica dos sintagmas nominais com quantificadores.

Para tal problema, a solução computacional proposta é o uso de *armazenamentos*. Nesta abordagem, a representação semântica de cada expressão deixa de ser a de uma simples fórmula em cálculo lambda, para ser a de uma representação de múltiplas formas possíveis.

Em particular, começaremos com o *armazenamento de Cooper*. Esta é uma técnica desenvolvida por Robin Cooper para ... (Blackburn and Bos, 2005, p. 113)

2.4 Armazenamento de Keller

2.5 *Hole Semantics*

3 ...

4 Conclusão

5 References

Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI, 2005.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. ISBN 0131873210.