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

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, não-ambigüidade, existência de uma forma canônica, capacidade de inferência, uso de variáveis e expressividade. Todas estas são possuídas pela lógica de primeira ordem, ao menos até certo ponto. Além disso, é um sistema bem compreendido e bastante flexível.

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.

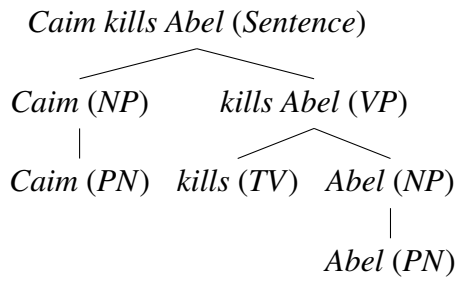
Por exemplo, podemos entender que o significado de "*kills*" é o predicado binário $kill(\ldots,\ldots)$, onde convencionamos que o primeiro argumento é o agressor (isto é, aquele que mata) e o segundo argumento é a vítima (aquele que é morto). Também podemos entender os significados de "*Caim*" e "*Abel*" 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 $kill(caim, abel)$.

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

$$
\begin{aligned}
&(\lambda k.\lambda y.k@(\lambda x.kill(y,x)))@\lambda u.u@abel\\
&\lambda y.((\lambda u.u@abel)@(\lambda x.kill(y,x)))\\
&\lambda y.(\lambda x.kill(y,x)@abel)\\
&\lambda y.kill(y,abel)
\end{aligned}
$$

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.

2.2.1 Dificuldades – Ambigüidades de Escopo

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*. (Blackburn and Bos, 2005, p. 105-109) As ambigüidades de escopo são melhor explicadas através de exemplos.

Analisemos a frase:

"*Every man loves a woman.*"

Esta frase parece ter duas interpretações possíveis: na primeira, para cada homem

existe uma mulher amada por aquele. Possivelmente, são mulheres distintas. Já na segunda leitura, existe uma mulher específica que é amada por todos os homens.

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.

Que o nosso sistema atual não é capaz de representar esse tipo de ambigüidade pode ser visto pelo fato de que a representação semântica é única, dados o sentido dos termos mais simples e a decomposição sintática. Precisamos, assim, aprimorar o modelo.

Para termos um olhar em direção à solução, podemos notar 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.

2.3 Armazenamento de Cooper

Para o problema das ambigüidades de escopo, 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 lidar com ambigüidades de escopo de quantificadores. (Blackburn and Bos, 2005, p. 113) Intuitivamente, a idéia está em adicionar a possibilidade de substituir uma representação mais detalhada de um sintagma nominal por uma nova variável e "armazenar" a representação completa deste sintagma nominal para uso posterior. Ao fim, as representações podem ser "resgatadas" do armazenamento, em qualquer ordem. Ao se "resgatar" uma representação após alguma outra, o quantificador do sintagma nominal resgatado posteriormente poderá ter escopo mais externo do que um

quantificador da representação "resgatada" anteriormente. Desse modo, ao se possibilitar os "resgates" em ordens distintas, diferentes representações são formadas.

Agora cada expressão (isto é, cada nó da árvore de análise sintática (*parse*)) é associada a uma n-upla chamada "armazenamento". O primeiro elemento do armazenamento será uma fórmula de cálculo lambda, bem como antes. É uma representação "nuclear" da expressão. Com efeito, chamaremos este elemento de *núcleo* do armazenamento. Por sua vez, os outros elementos da n-upla serão pares (β, i), em que β é uma representação semântica para um sintagma nominal e i é um indice para este sintagma. Estes pares são denominados *operadores de ligação indexados* (*indexed binding operators*).

Com mais detalhes, *a priori* as representações não diferem muito de como eram. Os nós das folhas, não sendo nenhum um sintagma nominal quantificado, são análogos ao modo anterior, sendo armazenamentos com apenas uma entrada. Já um nó não-terminal pode ter sua representação montada de um modo "usual": ele tem como núcleo uma combinação dos núcleos de cada um de seus filhos na árvore; isto é, é a combinação dos núcleos dos armazenamentos dos termos que compõem a expressão mais complexa. Esta combinação é exatamente do mesmo modo como era feito até então. O restante do armazenamento do nó não-terminal é a justaposição (*append*) do restante dos armazenamentos de cada um dos termos filhos. Em suma: quando a expressão é composta por outras na análise sintática, tudo ocorre de modo análogo a como ocorria na representação "pura" por cálculo lambda, preservando os operadores de ligação indexados de todas as sub-expressões que compõem a expressão maior.

Caso o nó não-terminal não seja um sintagma nominal quantificado, a represnetação "usual" é a sua única possível. Entretanto, o processo possui uma diferença quando o nó não-terminal é um sintagma nominal quantificado. Além da composição "usual" para outros nós, há uma segunda representação possível. Isso merece ser destacado:

Armazenagem (Cooper)

Seja o armazenamento $\langle \phi, (\beta, j), \ldots, (\beta', k) \rangle$ a representação semântica "usual" para um sintagma nominal quantificado. O armazenamento $\langle \lambda u.(u@z_i), (\phi, i), (\beta, j), \ldots, (\beta', k) \rangle$, onde i é um índice único[1] também é uma representação para este sintagma nominal quantificado.

[1] isto é, não utilizado até então

Isto significa que sintagmas nominais quantificados podem ter suas representações montadas de dois modos. Neste ponto, nosso algoritmo terá uma escolha de aplicar ou não a regra de armazenagem. Ao se desejar saber a representação de uma frase em específico, esperaremos que nosso sistema nos retorne todas as representações possíveis. Perceba também que a regra não é recursiva. Há apenas duas opções: manter a representação "usual" ou realizar a operação de armazenagem.

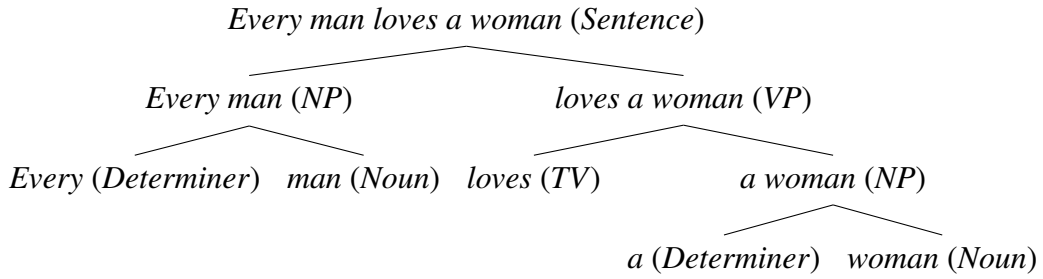
Após todo este processo, teremos uma frase cuja representação é um armazenamento. É necessário lidar com isto de algum modo, pois o que desejamos é que uma frase possa ser representada por expressões de lógica de primeira ordem, não por um armazenamento. Aqui é que poderemos "resgatar" nossos operadores de ligação indexado, que foram previamente armazenados. Para isso, usaremos a seguinte regra de resgate:

Resgate (Cooper)

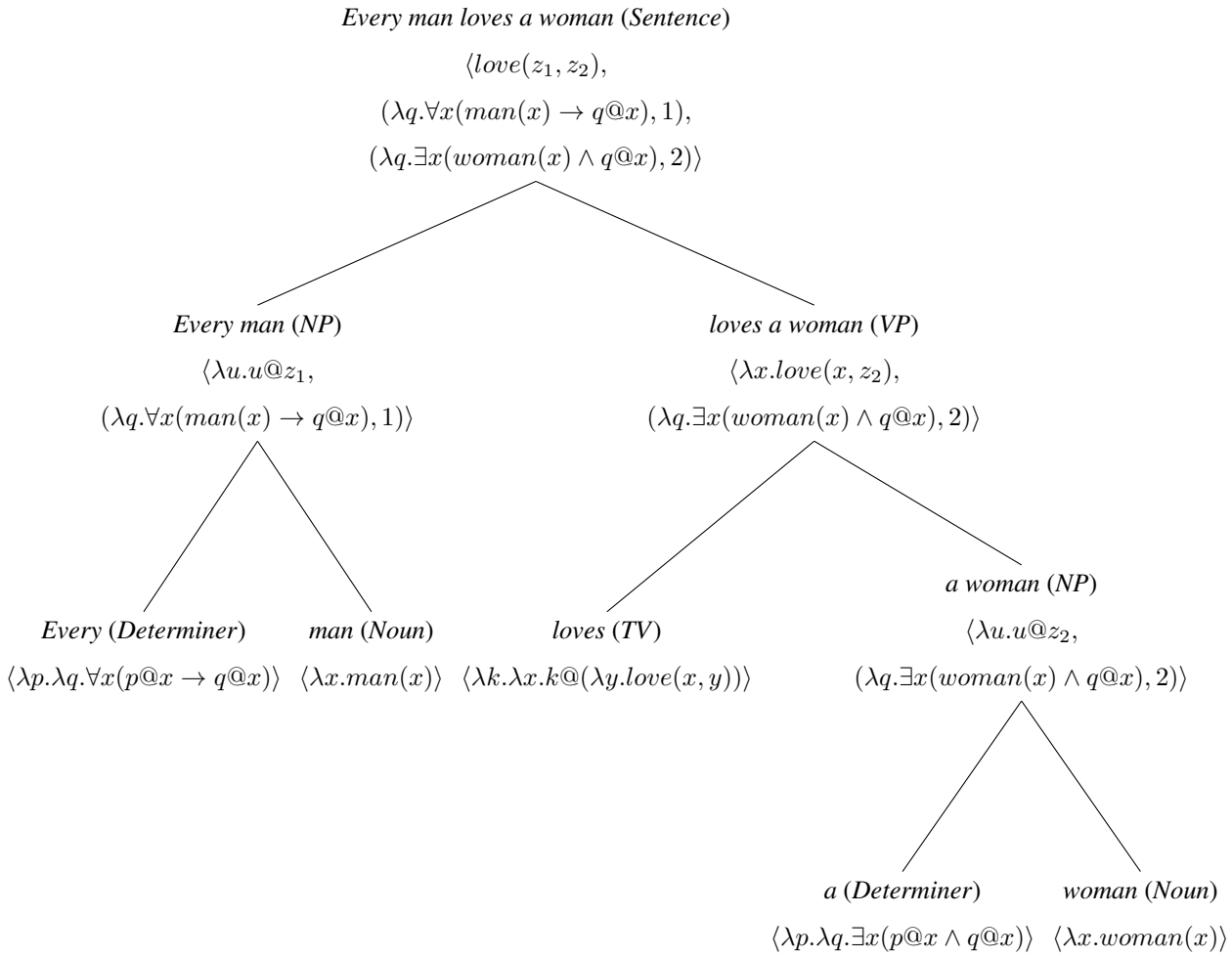
Sejam σ_1 e σ_2 duas seqüências (possivelmente vazias) de operadores de ligação. Se o armazenamento $\langle \phi, \sigma_1, (\beta, i), \sigma_2 \rangle$ a uma frase (*sentence*), então o armazenamento $\langle \beta @ \lambda z_i . \phi, \sigma_1, \sigma_2 \rangle$ também é associado a esta frase.

Um armazenamento composto apenas por um núcleo, após sucessivas aplicações da regra de resgate, será uma fórmula bem formada de primeira ordem, como desejávamos.

Para visualizarmos este processo, vamos para um exemplo:



Esta é a árvore de análise sintática. Construindo os significados a partir das folhas e subindo, uma das possíveis árvores que podemos alcançar é:



Agora, o que resta é converter o armazenamento representativo da frase completa nas possíveis fórmulas de primeira ordem através da operação de resgate.

Inicialmente:

$$\langle love(z_1, z_2),$$
$$(\lambda q.\forall x(man(x) \rightarrow q@x), 1), (\lambda q.\exists x(woman(x) \wedge q@x), 2)\rangle$$

Resgatando o operador de ligação 1:

$$\langle \lambda q.\forall x(man(x) \rightarrow q@x)@(\lambda z_1.love(z_1, z_2)),$$
$$(\lambda q.\exists x(woman(x) \wedge q@x), 2)\rangle$$

β-convertendo:

$$\langle \forall x(man(x) \rightarrow love(x, z_2)),$$
$$(\lambda q.\exists x(woman(x) \wedge q@x), 2)\rangle$$

Resgatando o operador de ligação 2:

$$\langle (\lambda q.\exists x(woman(x) \wedge q@x))@(\forall x(man(x) \rightarrow love(x, z_2)))\rangle$$

α-convertendo e β-convertendo:

$$\langle \exists x(woman(x) \wedge \forall y(man(y) \rightarrow love(y, x)))\rangle$$

Assim, chegamos a uma das duas interpretações: a de que todos os homens amam uma mesma mulher. Se resgatarmos o operador de ligação 2 e só depois resgatarmos o operador de ligação 1, teremos a outra leitura: $\forall y(man(y) \rightarrow \exists x(woman(x) \wedge love(y, x)))$

Portanto, desenvolvemos um método sistemático que pode capturar as ambigüidades de escopo, produzindo as leituras possíveis. O que nos resta agora é a pergunta: será que nosso método é de fato capaz de capturar todas as ambigüidades de escopo? Infelizmente, deve estar claro que não. Iremos apontar duas frases nas quais o método não é suficiente.

2.3.1 Dificuldades

A primeira frase é: "*Every criminal with a gun is dangerous.*" Aplicando nosso método, teremos os seguintes resultados:

1. $\forall x((criminal(x) \wedge \exists y(gun(y) \wedge with(x, y))) \rightarrow smoke(x))$
2. $\exists y(gun(y) \wedge \forall x(criminal(x) \wedge with(x, y) \rightarrow smoke(x)))$

3. $\forall x((criminal(x) \wedge with(x, y)) \rightarrow \exists z(gun(z) \wedge smoke(x)))$

Apesar dos resultados 1 e 2 serem perfeitamente razoáveis, sendo as interpretações que desejávamos, a interpretação 3 possui uma variável livre, não sendo uma sentença de primeira ordem. Isso nos mostra que há um problema com o nosso método. Como isso surgiu?

Realizando nosso procedimento e optando sempre por colocar a representação do sintagma nominal no armazenamento, montaremos a árvore:

Por sua vez, a segunda frase é: "*Every man doesn't love a woman*". A presença da negação traz elementos interessantes. Em primeiro lugar, ela em si é uma fonte possível de ambigüidades de escopo. Entretanto, o método de armazenamento de Cooper não tratou a negação de nenhum modo especial. Além disso, esse exemplo mostra o interesse em manter a operação de armazenamento como opcional. Esta frase pode ser interpretado de seis modos:

1. $\neg\forall x(man(x) \rightarrow \exists y(woman(y) \wedge love(x, y)))$
2. $\neg\exists y(woman(y) \wedge \forall x(man(x) \rightarrow love(x, y)))$
3. $\forall x(man(x) \rightarrow \neg\exists y(woman(y) \wedge love(x, y)))$
4. $\exists y(woman(y) \wedge \neg\forall x(man(x) \rightarrow love(x, y)))$
5. $\forall x(man(x) \rightarrow \exists y(woman(y) \wedge \neg love(x, y)))$
6. $\exists y(woman(y) \wedge \forall x(man(x) \rightarrow \neg love(x, y)))$

Apesar disso, nosso método apenas gerará três desses modos: 3, 5 e 6. Assim, a presença da negação de fato afeta a capacidade de nosso sistema produzir todas as interpretações.

2.4 Armazenamento de Keller

Para lidar especificamente com o primeiro problema do armazenamento de Cooper, Bill Keller propôs uma alteração: permitir armazenamentos aninhados. Assim, cada operador de ligação passa a ser composto não mais por uma fórmula de cálculo lambda e um índice único, mas sim por um armazenamento e um índice único. Isto altera a regra de armazenagem:

Armazenagem (Keller)

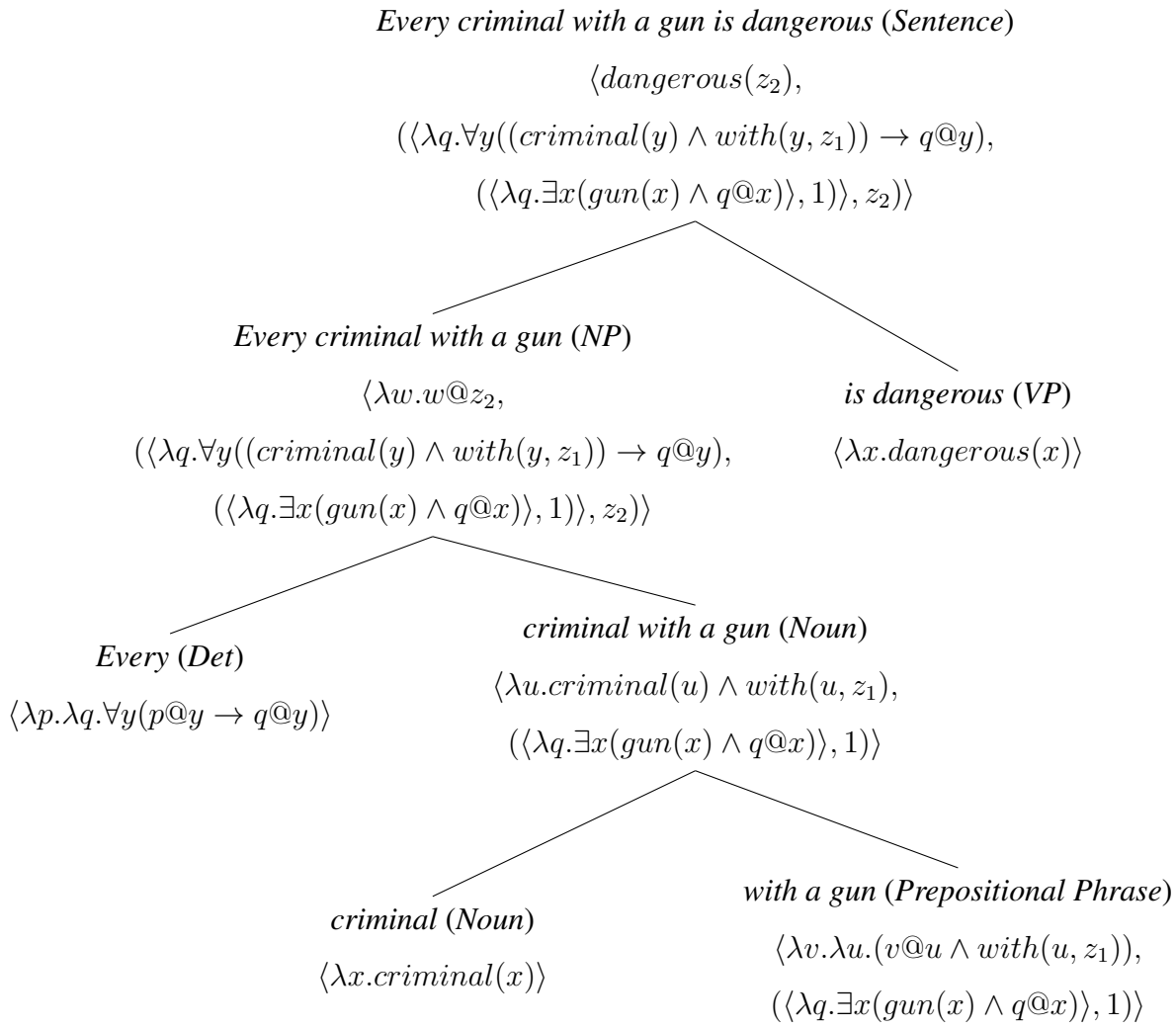
Sendo σ uma seqüência (possivelmente vazia) de operadores de ligação, se o armazenamento $\langle \phi, \sigma \rangle$ é a representação semântica "usual" para um sintagma nominal quantificado, então o armazenamento $\langle \lambda u.(u@z_i), (\langle \phi, \sigma \rangle, i) \rangle$, onde i é um índice único, também é uma representação para este sintagma nominal quantificado.

Por sua vez, também o resgate é alterado. Um operador de ligação só pode ser resgatado para aplicação do núcleo do armazenamento se todos os armazenamentos externos a ele já tiverem sido aplicados. Isto garante que, caso os sintagmas nominais estejam aninhados, então o sintagma nominal mais interno só terá seu operador resgatado após o resgate do sintagma nominal mais externo, evitando o tipo de problema que observamos. Portanto, nossa regra é:

Resgate (Keller)

Sejam σ, σ_1 e σ_2 seqüências (possivelmente vazias) de operadores de ligação. Se o armazenamento $\langle \phi, \sigma_1, ((\beta, \sigma), i), \sigma_2 \rangle$ é uma representação para uma frase (*sentence*), então $\langle (\beta @ \lambda z_i.\phi), \sigma_1, \sigma, \sigma_2 \rangle$ também o é.

Podemos então aplicar isto para o nosso exemplo:



Agora, realizando a extração, podemos fazê-la apenas de um modo: $\exists x(gun(x) \wedge \forall y((criminal(y) \wedge with(y, x)) \rightarrow dangerous(y)))$. Isto é a interpretação correta, não tendo sido gerado nenhum problema. As outras opções de (não-)extração funcionam de modo semelhante.

Assim, o problema dos sintagmas nominais é resolvido. Apesar disso, o segundo problema apontado, do escopo das negações, persiste. As interpretações geradas são as mesmas de antes, pelo armazenamento de Cooper. Portanto, o método de Keller aprimora os resultados de Cooper, sem resolver todas os obstáculos gerados por ambigüidades de escopo.

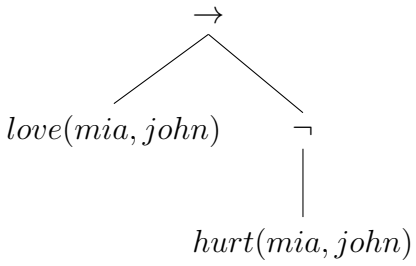
2.5 *Hole Semantics*

Apesar de ser possível criar um novo mecanismo para capturar a ambigüidade de escopo gerada pela negação , abordagens *ad hoc* para novas dificuldades não são muito

desejadas, criando uma falta de harmonização dos métodos usados, possivelmente proliferando uma diversidade de construções muito distintas entre si. Se possível, gostaríamos de possuir uma abordagem mais uniforme. Na realidade, não apenas a negação traz ambigüidades de escopo. Por exemplo, uma frase como "*If a man walks then he jumps and a woman is happy*" é ambígua. Bos (1996) Podemos imaginar uma interpretação na qual "*a woman is happy*" é parte do conseqüente da implicação e outra na qual não o é, sendo uma afirmação separada da implicação.[2] Em razão destas dificuldades, e também de modo a ganhar maior flexibilidade na representação, analisaremos uma outra forma de representação semântica, não baseada em armazenamentos.

Assim como nos métodos baseados em armazenamentos, uma frase não será associada uma expressão de primeira ordem, mas a uma representação abstrata, que então é associada a um conjunto de expressões em primeira ordem. Entretanto, o modo pelo qual isso é feito aqui é distinto. Em *Hole Semantics*, uma idéia essencial é a de *restrições*: podemos pensar na representação como um conjunto de restrições, de modo que qualquer fórmula de primeira ordem que satisfaça as restrições será uma interpretação possível para a frase. (Blackburn and Bos, 2005, p. 129) A representação será referida por representação subespecificada (*USR*, de *underspecified representation*).

Uma fórmula de primeira ordem possui uma decomposição única como uma árvore, em razão pelo modo como é montada. Por exemplo, para a frase "*If Mia loves John then Mia does not hurt John*" tem associada à sua semântica a fórmula $love(mia, john) \rightarrow (\neg hurt(mia, john))$, que pode ser decomposta na árvore:



As restrições serão sobre o modo de construir a fórmula. Dito de outro modo, a USR será um modo de falar a respeito da árvore de cada interpretação possível. Ao invés de montarmos uma única árvore (o que corresponderia a uma única fórmula), a USR pode ser pensada como uma "árvore incompleta", isto é, uma árvore com "buracos", justificando o nome desse método. Entretanto, estes buracos não poderão ser preenchidos de qualquer

[2]Esta construção não ocorre em nosso programa, por não haver orações coordenadas com "*and*".

modo, havendo relações de *dominância*. Um buraco deverá dominar um nó quando estiver acima do mesmo na representação da árvore. A partir daí, as subfórmulas irão compôr a fórmula completa através de um "preenchimento" dos buracos. Este "preenchimento" será feito *encaixando* algum nó (junto com sua sub-árvore) no buraco.

Nos métodos de armazenamentos, a semântica das frases era representada por um vetor que continha um núcleo e os quantificadores guardados em (um aninhamento de) operadores de ligação. Em *Hole Semantics*, nosso modo de representar será bem distinto. Na realidade, nós usaremos uma linguagem lógica para essa representação, chamada *linguagem de representação subespecificada* (*URL*, do inglês *underspecified representation language*). A linguagem original, que aqui é alguma forma de lógica de primeira ordem, será referida por *linguagem de representação semântica* (*SRL*, *semantic representation language*). Pode causar algum espanto o fato de que a URL será, ela própria, uma linguagem de primeira ordem! Seu vocabulário será definido do seguinte modo:

1. Predicados binários :NOT e $\leq$

2. Predicados ternários :IMP, :AND, :OR, :ALL, :SOME e :EQ

3. Cada constante no vocabulário da SRL também é uma constante no vocabulário da URL.

4. Para cada predicado n-ário $pred$ na SRL, :PRED é um predicado $(n+1)$-ário na URL.

A lógica de primeira ordem utilizada é *tipada*, havendo três tipos. O primeiro deles é o dos *buracos*, cujas variáveis serão denotadas por h, h', h_1, h_2, etc. O segundo é o tipo dos *rótulos*, cujas variáveis são escritas l, l', l_1, etc. Cada rótulo marcará um vértice na árvore que não é um buraco, sendo um modo de se referir aos símbolos da SRL. Por fim, o terceiro tipo é o das *meta-variáveis*, escritos v, v', v_1, v_2, etc. As meta-variáveis têm a função de se referir às variáveis da SRL.

Dizemos que algo é um *nó* se for um buraco ou um rótulo. Dizemos que algo é um *meta-termo* da URL caso seja uma *meta-variável* ou uma constante da URL.

Agora, iremos definir as USRs básicas:

1. Se l é um rótulo e h é um buraco, então $l \leq h$ é uma USR básica.

2. Se l é um rótulo e n e n' são nós, então l:NOT(n), l:IMP(n, n'), l:AND(n, n') e l:OR(n, n') são USRs básicas.

3. Se l é um rótulo enquanto t e t' são meta-termos, então l:EQ(t, t') é uma USR básica.

4. Se l é um rótulo, S é um símbolo n-ário na linguagem SRL e $t_1, \ldots, t_n$ são meta-termos, então l:S$(t_1, \ldots, t_n)$ é uma USR básica.

5. Se l é um rótulo, v é uma meta-variável, n é um nó, então l:SOME(v, n) e l:ALL(v, n) são USR básicas.

6. Nada mais é uma USR.

Observe aqui que o espaço a mais criado pela subida de aridade nos predicados e conectivos é preenchido pela variáveis de rótulo. Observe que o item **??** é o único que utiliza o símbolo de $\leq$. USRs básicas desta forma são ditas *restrições de dominância*. Por fim, podemos definir o restante das USRs:

1. Toda USR básica é uma USR.

2. Se ϕ é uma USR e n é um nó, então $\exists n\phi$ é uma USR.

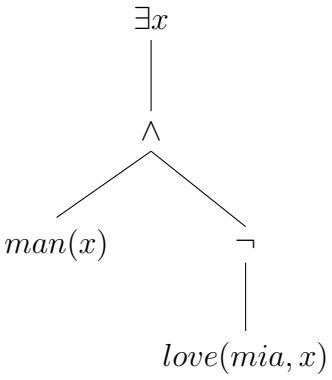
3. Se ϕ é uma USR e v é uma meta-variável, então $\exists v\phi$ é uma USR.

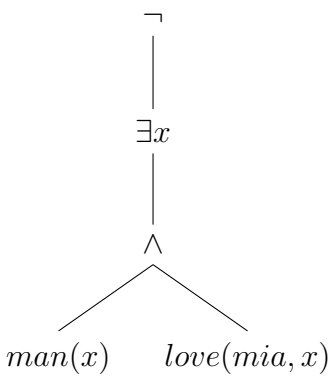
4. se ϕ e ψ são USRs, então $\phi \wedge \psi$ é uma USR.

5. Nada mais é uma USR.

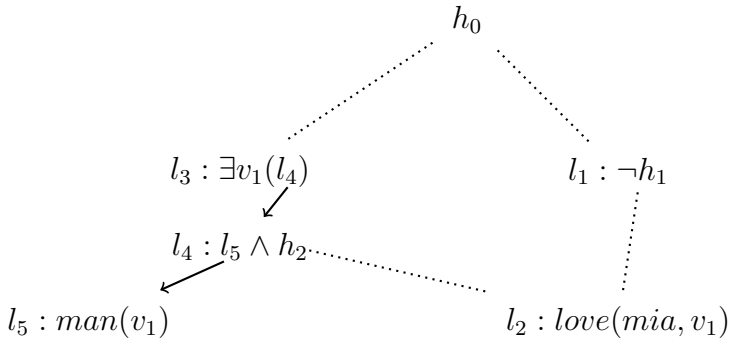
É de ser notado que nem todos os conectivos e formas da lógica da primeira ordem foram empregados nesta definição. Na realidade, apenas são fórmulas conjuntos pequenos de formas conjuntivas e existencialmente fechadas. Entretanto, esse fragmento da linguagem é suficiente para nossos propósitos. (Blackburn and Bos, 2005, p. 131)

Podemos avançar então para um exemplo. Consideremos a frase "*Mia does not love a man*". Uma interpretação é aquela na qual Mia não ama um homem específico, que pode ser formalizada como $\exists x : man(x) \wedge \neg love(mia, x)$. Outra, é aquela na qual Mia não ama homem algum, isto é, $\neg(\exists x : man(x) \wedge love(mia, x))$. Suas árvores são:





Já a representação subespecificada busca capturar o que há em comum entre as árvores possíveis. A USR desta frase é: $\exists h_0 \exists h_1 \exists h_2 \exists l_1 \exists l_2 \exists l_3 \exists l_4 \exists l_5 \exists v_1 (l_1 : \text{NOT}(h_1) \wedge l_2{:}\text{LOVE}(mia, v_1) \wedge l_3{:}\text{SOME}(v_1, l_4) \wedge l_4{:}\text{AND}(l_5, h_2) \wedge l_5{:}\text{MAN}(v_1))$. Porém, as USRs se tornam melhor compreensíveis através de sua representação gráfica, estando abaixo aquela relativa à nossa frase considerada:



Podemos ver a intuição desta representação. É criado um buraco h_0 correspondente ao nó mais alto da árvore. As linhas pontilhadas representam restrições de dominância entre buracos e nós. Por sua vez, as linhas preenchidas mostram quais nós são pais de outros. A relação de parentesco também representa dominância: se um nó é pai de outro, é certo que o filho não pode ter escopo mais externo que o pai, uma vez que deve ser subfórmula do mesmo. Entretanto, neste caso a posição está fixa: necessariamente a relação de parentesco será aquela. Por sua vez, na dominância entre buracos e nós, não é isto que occore. Basta que o nó dominado esteja no escopo do nó dominante, não necessariamente sendo filho do mesmo. Ou seja, basta ser descendente.

Agora, a nossa análise de frases é feita ainda decompondo sintaticamente, e então, para cada termo, criando uma representação na forma de uma USR. Ainda utilizamos o cálculo lambda para fazer combinações de expressões. A representação final da frase é

feita por combinações das representações das partes que as constituem. Por exemplo, a representação para o determinante "*a*" é: $\lambda x.\lambda y.\lambda h.\lambda l.\exists h_1 \exists l_1 \exists l_2 \exists l_3 \exists v_1 (l2{:}\textsc{all}(v_1, l_3, \wedge l3{:}\textsc{and}(l_1, h_1) \wedge l \leq h_1 \wedge l_2 \leq h \wedge x@v_1@h@l_1 \wedge y@v_1@h@l))$

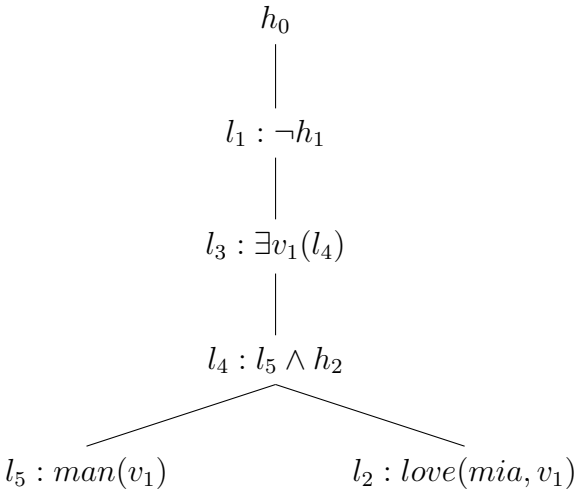
Por sua vez, para o substantivo "*woman*" é: $\lambda v.\lambda h.\lambda l.(l{:}\textsc{woman}(v) \wedge l \leq h)$

Assim, o sintagma nominal *"a woman"* fica, aplicando a segunda representação na primeira e beta-reduzindo: $\lambda y.\lambda h.\lambda l.\exists h_1 \exists l_1 \exists l_2 \exists l_3 \exists v_1 (l2{:}\textsc{all}(v_1, l_3, \wedge l3{:}\textsc{and}(l_1, h_1) \wedge l \leq h_1 \wedge l_2 \leq h \wedge l_1{:}\textsc{woman}(v_1) \wedge l_1 \leq h \wedge y@v_1@h@l))$.

Definindo as USRs para cada função sintática e o modo de combiná-las, obtemos a USR da frase. Com isto em mãos, precisamos ser capazes de construir as árvores possíveis. Isso é feito por meio de *encaixes*[3]. Para cada buraco, achamos um rótulo candidato para preenchê-lo: este rótulo será encaixado no buraco. Mais formalmente, um encaixe é uma função injetiva dos buracos aos rótulos. Entretanto, nem todo encaixe nos satisfaz. Evidentemente, queremos satisfazer duas condições: queremos que o resultados seja uma árvore (portanto, acíclica e conexa), bem como queremos que, se existe uma restrição de dominância de um buraco H sobre um rótulo L (ou seja, se $L \leq H$), então L será descendente de H na árvore.

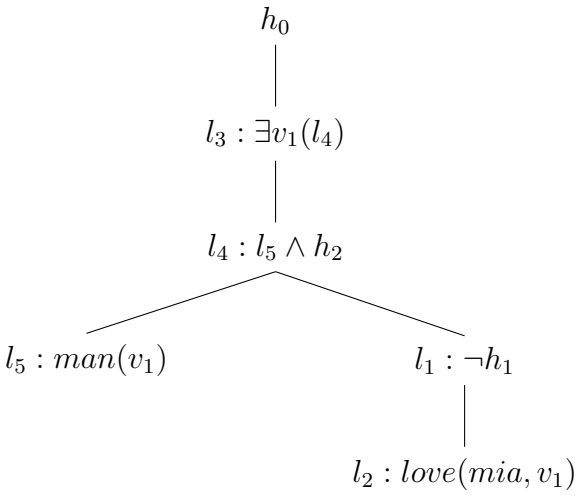
Para o exemplo que vimos, dois encaixes são possíveis: $P_1(h_0) = l_1, P_1(h_1) = l_3, P_1(h_2) = l_2$ e $P_2(h_0) = l_3, P_2(h_1) = l_2, P_2(h_2) = l_1$.

Assim, duas árvores são formadas, cada uma gerando uma interpretação possível. Pelo encaixe P_1 obtemos:



Já pelo P_2, temos:

[3]Em inglês, o termo usado é "*plug*", por isso a letra utilizada é P.



Com efeito, estas são de fato as árvores que havíamos construído antes, para cada interpretação.

3 ...

4 Conclusão

5 References

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

Johan Bos. Predicate logic unplugged. In *In Proceedings of the 10th Amsterdam Colloquium*, pages 133–143, 1996.

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