Análisis sintáctico de la lógica de primer orden

Emmanuel Peto Gutiérrez

IIMAS
UNAM

5 de diciembre de 2022

Razonamiento automatizado

El razonamiento automatizado es el área de las ciencias de la computación concentrada en aplicar el razonamiento de forma lógica a un sistema.
Dados un conjunto de hipótesis y una meta, un sistema de razonamiento automatizado debería hacer inferencias lógicas para llegar a la meta automáticamente.

Razonamiento automatizado

El razonamiento automatizado es el área de las ciencias de la computación concentrada en aplicar el razonamiento de forma lógica a un sistema.
Dados un conjunto de hipótesis y una meta, un sistema de razonamiento automatizado debería hacer inferencias lógicas para llegar a la meta automáticamente.
Si hago un programa que haga inferencia, el primer problema a resolver es el de darle la entrada al programa.
¿Cuál es el formato que entiende mi programa?

Gramática en lógica de primer orden

Lenguajes como Haskell nos permiten definir gramáticas.
Lo que entendemos nosotros:

$t ::= x | f[t]$

$at ::= \top | \bot | P[t] | t = t$

$\star ::= \rightarrow | \leftrightarrow | \wedge | \vee$

$Q ::= \forall | \exists$

$f ::= at | f \star f | Qxf$

Gramática en lógica de primer orden

Lenguajes como Haskell nos permiten definir gramáticas.
Lo que entendemos nosotros:
$t ::= x | f[t]$
$at ::= \top | \bot | P[t] | t = t$
$\star ::= \rightarrow | \leftrightarrow | \wedge | \vee$
$Q ::= \forall | \exists$
$f ::= at | f \star f | Qxf$
Lo que entiende Haskell:

```
data Term = Var String | Fun String [Term]
data Form = Top | Bot | Pred String [Term] | Eq
Term Term | Neg Form | Or Form Form | And Form
Form | Impl Form Form | Syss Form Form | Forall
String Form | Exists String Form
```

Parser

Problema: construir un programa que lea cadenas de texto (sintaxis concreta) y lo transforme a una expresión que sea entendible por Haskell (sintaxis abstracta).

Traducir de sintaxis concreta a sintaxis abstracta es un tema bien conocido porque es de central importancia en lenguajes de programación. Es convencional separar la transformación en dos pasos:

Traducir de sintaxis concreta a sintaxis abstracta es un tema bien conocido porque es de central importancia en lenguajes de programación. Es convencional separar la transformación en dos pasos:

- **Análisis léxico:** descomponer el texto de entrada en "tokens".

Traducir de sintaxis concreta a sintaxis abstracta es un tema bien conocido porque es de central importancia en lenguajes de programación. Es convencional separar la transformación en dos pasos:

- **Análisis léxico:** descomponer el texto de entrada en "tokens".
- **Análisis sintáctico:** convertir la sucesión de tokens en un árbol de sintaxis abstracta.

Términos

El primer paso es construir un parser de términos.
Contrato:

- Si al término lo precede una ‘c’, es constante.
- Si lo precede una ‘v’, es variable.
- Si lo precede una ‘f’, es función.

La función que realiza el análisis sintáctico de un término se llama `parseTerm`.
Ejemplo:

```
OK, two modules loaded.
*Parser> parseTerm "ff(cc,vx)"
f(c,x)
*Parser>
```

Tokens

El primer paso para analizar una fórmula es separar el texto en tokens, donde un token será un par (`tipo`, `token`). Es decir, cada token estará unido con su tipo.
Los diferentes tipos de tokens son: cuantificador, paréntesis izquierdo, paréntesis derecho, predicado, conector.
Ejemplo:

```
*Parser> tokenizer "∀x[P(x)→Q(x)]"
[("cuantificador","\8704x"),("(","("),("predicado","P(x)"),("conector","\8594"),
("predicado","Q(x)"),(")",")")]
```

Notación postfija

El segundo paso es transformar los tokens de notación infija a postfija. En este paso se eliminarán los paréntesis.
Ejemplo:

```
*Parser> infApost2 "∀x[P(x)→Q(x)]"
[("predicado","P(x)"),("predicado","Q(x)"),("conector","\8594"),
("cuantificador","\8704x")]
*Parser>
```

Sintaxis abstracta

En el último paso, se toma una expresión de tokens en notación postfija y se transforma a una fórmula de Haskell (definida con `data Form`).
Ejemplo:

```
*Parser> parseForm "∀x[P(vx)→Q(vx)]"
∀x[P(x) → Q(x)]
```

Alfa-equivalencias

Uno de los problemas que surgen en la representación con nombres es que dos fórmulas pueden ser distintas aunque el significado sea el mismo. Cuando esto ocurre, se dice que son α-*equivalentes*.

Ejemplo, las siguientes fórmulas son α-equivalentes: $\forall x P(x)$ y $\forall y P(y)$.

Si le preguntamos a Haskell si las fórmulas son iguales nos va a decir que no (imprime False).

```
*Parser> let form1 = parseForm "∀xP(vx)"
*Parser> let form2 = parseForm "∀yP(vy)"
*Parser> form1
∀xP(x)
*Parser> form2
∀yP(y)
*Parser> form1 == form2
False
```

Representación local sin nombres

Se pueden representar las fórmulas y términos con una gramática diferente, la cual quita los nombres de las variables ligadas y pone números. A esos números se les conoce como índices de de Bruijn.
Una variable con índice i apunta al $(i+1)$-ésimo cuantificador que lo encierra. Además, la gramática hace una diferencia explícita entre las variables ligadas (`Bvar`) y las libres (`Fvar`). La nueva gramática es la siguiente, donde t son los términos y *form* las fórmulas.

- $t ::=$ Bvar i | Fvar *name* | f *name* [t]
- *form* $::= \forall form | \exists form$ ...

Nota: los puntos suspensivos en *form* significan que el resto de la gramática se queda igual.

Transformación de fórmulas

Se realizó una función que transforma fórmulas *crudas* a su equivalente en representación local sin nombres.
Si se tienen dos fórmulas α-equivalentes tal que no tienen variables libres, entonces se verán exactamente igual en su representación sin nombres.
Ejemplo:

```
*Parser> let form3 = transFormLNR $ parseForm "∀xP(vx)"
*Parser> let form4 = transFormLNR $ parseForm "∀yP(vy)"
*Parser> form3
∀P(0)
*Parser> form4
∀P(0)
*Parser> form3 == form4
True
```

FIN