Análisis sintáctico en la lógica de primer orden

Emmanuel Peto Gutiérrez

6 de diciembre de 2022

Resumen. En este artículo se hablará sobre cómo implementar un analizador sintáctico de lógica de primer orden en el lenguaje de programación Haskell. Después se mencionarán diferentes gramáticas usadas para representar variables ligadas, y entre esas se mostrará la representación local sin nombres. Finalmente se hablará sobre cómo transformar una expresión formada con una gramática que no distingue entre variables ligadas y libres (gramática cruda) a una con representación local sin nombres.

1. Introducción

Traducir de sintaxis concreta a sintaxis abstracta es un tema bien conocido porque es de central importancia en compiladores e intérpretes de lenguajes de programación y en traductores. Es convencional separar la transformación en dos partes[1]:

- Análisis léxico (escaneo): descompone la secuencia de caracteres de entrada en "tokens" (hablando de manera informal, palabras). parsing converts the linear sequences of tokens into an abstract syntax tree.
- Análisis sintáctico: convierte la secuencia lineal de tokens en un árbol de sintaxis abstracta.

Por ejemplo, el análisis léxico podría separar la entrada "v10 + v11" en tres tokens: "v10", "+" y "v11", juntando caracteres alfanuméricos en palabras y desechando cualquier número de espacios entre esos tokens. Después, el análisis sintáctico tiene que lidiar solo con secuencias de tokens y puede ignorar detalles de bajo nivel.

2. Análisis de términos

En esta sección se hablará de la forma de hacer análisis sintáctico sobre términos.

2.1. Gramática de términos

La primera tarea es definir un tipo de dato que representa la sintaxis abstracta de términos. Un término se puede definir con la siguiente gramática (o notación de Backus-Naur).

$t ::= x | f(t_1, t_2, ..., t_n)$

Lo que significa que un término es o una variable (x) o es una función (f) que tiene términos como parámetros. Si la función recibe 0 parámetros se dirá que es una *constante*.

Un dato completamente nuevo se puede declarar especificando sus valores usando el mecanismo `data` de Haskell[2]. Se definen los términos en Haskell de la siguiente forma:

```
data Term = Var String | Fun String [Term]
```

`Var` se usará para representar una variable y el `String` será para darle nombre a esa variable. Una función se define con `Fun`, el `String` es para el nombre de la función y también tiene una lista de términos (`[Term]`).

2.2. Sintaxis concreta de términos

Para que el parser pueda transformar el texto (sintaxis concreta) a sintaxis abstracta, el texto debe cumplir con ciertas características. Para el parser implementado en este artículo he adoptado la convención de que un nombre (de función, variable o constante) tiene exactamente un caracter.

Un problema que surge al leer un término es que no hay una forma de diferenciar entre una variable y una constante. Para este parser he decidido usar la siguiente convención:

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

Así que, en sintaxis concreta, un nombre de término debe tener exactamente dos letras: c, v o f, seguido del caracter que representa su nombre. Por ejemplo, para representar al término $f(x, a)$ donde x es variable y a es constante, se debe escribir: `ff(vx,ca)`.

2.3. Funciones auxiliares

Antes de hablar del parser, se hablará de las funciones necesarias para implementarlo.

Se sabe que una función (como término) tiene entre paréntesis una lista de términos que recibe. Así que se necesita una función que tome todo lo que está dentro de los paréntesis y lo separe del resto del texto. Por ejemplo, si tengo la cadena (que está leyendo el parser) "(fg(cc),vy,ca),ce,cd,vx)", sería deseable separar entre "(fg(cc),vy,ca)" y ",ce,cd,vx)", sabiendo que "fg(cc),vy,ca" son los

parámetros de alguna función. La función `lastParen` realiza esa separación, usando como auxiliar la función `lastParen'`

```
lastParen::String->(String,String)
lastParen cadena = if (head cadena) == '('
                   then lastParen' (tail cadena) "(" 1
                   else ("empty","empty")

lastParen'::String->String->Int->(String,String)
lastParen' str1 str2 0 = (str2, str1)
lastParen' (caracter:resto1) str2 n = case caracter of
    '(' -> lastParen' resto1 (str2++"(") (n+1)
    ')' -> lastParen' resto1 (str2++")") (n-1)
    otherwise -> lastParen' resto1 (str2++[caracter]) n
```

Luego, si se tiene una secuencia de términos separados por comas, por ejemplo "fg(cc),vy,ca", se necesita separar cada término: "fg(cc)", "vy" y "ca". Esto es necesario para separar los parámetros de una función. La función `separaTerminos` realiza esta acción.

```
separaTerminos::String->[String]
separaTerminos "" = []
separaTerminos (',':cs) = separaTerminos cs
separaTerminos ('v':cs) = ('v':[(head cs)]) : (separaTerminos (tail cs))
separaTerminos ('c':cs) = ('c':[(head cs)]) : (separaTerminos (tail cs))
separaTerminos ('f':cs) = let par = lastParen (tail cs)
    in [("f"++[(head cs)]++(fst par))] ++ (separaTerminos (snd par))
```

2.4. Implementando el parser de términos

Una vez que se tienen estas funciones, se puede implementar el parser de términos, que transforma cadena (String) a un dato de tipo `Term`. La función `parseTerm` realiza esta transformación, usando como auxiliar la función `parseTermComas`, la cual simplemente separa una cadena de términos separados por comas y aplica la función `parseTerm` a cada uno de ellos.

```
parseTerm::String->Term
parseTerm "" = error ("Un término no puede ser cadena vacía")
parseTerm (c:cs) = case c of
    'c' -> Fun cs []
    'v' -> Var cs
    'f' -> Fun [head cs] (parseTermComas (init $ tail $ tail cs))

parseTermComas::String->[Term]
parseTermComas cadena = map parseTerm (separaTerminos cadena)
```

Como ejemplo, la función `parseTerm` recibe la cadena "ff(vx,ca)" y la transforma a: `Fun ``f'' [Var ``x'', Fun ``a'' []]`.

3. Análisis de fórmulas

Una vez que se tiene una función para hacer parse sobre términos, se puede empezar a estudiar el análisis léxico y sintáctico en fórmulas.

3.1. Gramáticas de fórmulas

La gramática para construir las fórmulas es la siguiente:
$f ::= \top|\bot|P(t_1, t_2, ..., t_n)|t_1 = t_2|\forall x f|\exists f|f \wedge f|f \vee f|f \rightarrow f|f \leftrightarrow f|\neg f$
Donde P es un predicado que recibe n términos.
La definición en Haskell es la siguiente:

```
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
```

3.2. Análisis léxico de fórmulas

El primer paso para transformar una fórmula de sintaxis concreta a sintaxis abstracta es separar el texto en tokens. En este caso, un token va a ir pegado con su tipo, así que se van a crear pares `(tipo, token)`. Los diferentes tipos de token pueden ser:

- Cuantificador
- Paréntesis izquierdo
- Paréntesis derecho
- Conector
- Predicado

Se necesitan predicados (en este caso, funciones de Haskell) que reciban un caracter y decidan si es un cuantificador, conector, predicado o paréntesis. Las siguientes funciones identifican cuantificadores y conectores.

```
esCuantif::Char->Bool
esCuantif ∀ = True
esCuantif ∃ = True
esCuantif _ = False

esConector::Char->Bool
esConector ¬ = True
esConector ∧ = True
esConector ∨ = True
esConector → = True
esConector ↔ = True
esConector _ = False
```

Se considera que si el caracter no es cuantificador ni paréntesis ni conector, entonces es un nombre de predicado. La siguiente función realiza la separación por tokens:

```
tokenizer::String->[(String,String)]
tokenizer "" = []
tokenizer (x:xs)
    | esCuantif x = ("cuantificador", [x,head xs]) : (tokenizer (tail xs))
    | esConector x = ("conector", [x]) : (tokenizer xs)
    | x == '(' = ("(", "(") : (tokenizer xs)
    | x == '[' = ("(", "(") : (tokenizer xs)
    | x == ')' = (")", ")") : (tokenizer xs)
    | x == ']' = (")", ")") : (tokenizer xs)
    | otherwise = let par = lastParen xs
                  in ("predicado", x:(fst par)) : (tokenizer (snd par))
```

3.3. Notación infija a postfija

En la notación infija se coloca una fórmula, seguido de un conector, seguido de otra fórmula: $F \star F$, donde $\star \in \{\rightarrow, \wedge, \vee, \leftrightarrow\}$. O si se usan cuantificadores, se coloca primero el cuantificador con la variable que está ligando, seguido de la fórmula: QxF. El caso de la negación es análogo al del cuantificador. Esta notación es la que se usa en la sintaxis concreta y es en ese orden en el que se encuentran los tokens.

En la notación postfija se colocan primero las dos fórmulas y después el conector: $FF\star$. En el caso de cuantificadores (o negación), se coloca primero la fórmula y después el cuantificador (o símbolo de negación). Como ejemplo, se tiene la siguiente fórmula en las dos representaciones.

- Infija: $\forall x[\exists y[P(x) \rightarrow Q(y)]]$
- Postfija: $P(x)Q(y) \rightarrow \exists y \forall x$

La ventaja de usar notación postfija es que se eliminan los paréntesis y es más fácil construir un árbol de sintaxis abstracta a partir de esta notación, usando una pila como estructura de datos auxiliar.

En la sección **The Stack ADT** de [3] se puede encontrar un algoritmo que transforma una expresión aritmética de notación infija a postfija. El algoritmo usado aquí se basa en ese, excepto que aquí se usa para fórmulas y no se reconoce precedencia de operadores.

El algoritmo tiene una subrutina en la que, si encuentra un paréntesis derecho, se extraen los elementos de la pila hasta que encuentra un paréntesis izquierdo. Así que necesito una función que tome una pila y una expresión postfija, y separe los elementos en dos: la expresión postfija más los elementos que quedan fuera de la pila y los que quedan en la pila. La siguiente función realiza esa separación.

```
extraePila :: [(String,String)] -> [(String,String)]
-> ([(String,String)],[(String,String)])
extraePila [] expr = ([],expr)
extraePila (x:xs) expr = if (fst x) == "("
                         then ((x:xs), expr)
                         else extraePila xs (expr++[x])
```

La siguiente función realiza la conversión de notación infija a postfija usando recursión de cola.

```
infApostCola :: [(String,String)] -> [(String,String)] -> [(String,String)] -> [(String,String)]
infApostCola [] [] exprSal = exprSal
infApostCola [] (x:xs) exprSal = infApostCola [] xs (exprSal++[x])
infApostCola (("(","("):xs) p exprSal = infApostCola xs (("(","("):p) exprSal
infApostCola ((con, "¬"):xs) p exprSal = infApostCola xs ((con, "¬"):p) exprSal
infApostCola (("cuantificador", q):xs) p exprSal = infApostCola xs (("cuantificador", q):p) exprSal
infApostCola ((")",")"):xs) p exprSal = let tupla = extraePila p exprSal
    in infApostCola xs (tail (fst tupla)) (snd tupla)
infApostCola (x:xs) p exprSal = if (fst x) == "conector"
    then
        let tupla = extraePila p exprSal
        in infApostCola xs (x:(fst tupla)) (snd tupla)
    else infApostCola xs p (exprSal++[x])
```

La función `infApost` simplemente llama a `infApostCola` donde, tanto la pila como la expresión postfija, son inicialmente vacías. Luego, la función `infApost2` simplemente realiza una composición de funciones: primero aplica el `tokenizer` (dividir el texto en tokens) y luego `infApost` (convertir de notación infija a postfija).

```
infApost :: [(String, String)] -> [(String, String)]
infApost expr = infApostCola expr [] []

infApost2 :: String -> [(String, String)]
infApost2 expr = infApost (tokenizer expr)
```

3.4. Construcción del árbol de sintaxis abstracta

Una vez que se tienen los tokens en notación infija se puede construir la expresión `Form`. Primero se definirán dos funciones: una recibe dos fórmulas y un conector en String y construye la fórmula que le corresponde a ese conector. Otra recibe un cuantificador, una fórmula y construye una fórmula con ese cuantificador.

```
   creaForm :: String ->Form ->Form ->Form
creaForm op f1 f2 = case op of
∧ ->And f1 f2
∨ ->Or f1 f2
→ ->Impl f1 f2
```

```
↔ ->Syss f1 f2

   creaFormCuanti :: String ->Form ->Form
creaFormCuanti (∀:cs) f = Forall cs f
creaFormCuanti (∃:cs) f = Exists cs f
```

La siguiente función construye un árbol de sintaxis abstracta dada una expresión separada en tokens y en notación postfija. Utiliza una pila para hacer la construcción.

```
parseForm' :: [(String,String)] -> [Form] -> Form
parseForm' [] p = head p
parseForm' (x:xs) p = if (fst x) == "conector"
    then
        if (snd x) == "¬"
        then let f = head p
                 p2 = tail p
             in parseForm' xs ((Neg f):p2)
        else let f1 = head (tail p)
                 f2 = head p
                 p2 = tail (tail p)
             in parseForm' xs ((creaForm (snd x) f1 f2):p2)
    else if (fst x) == "cuantificador"
         then let f = head p
                  p2 = tail p
              in parseForm' xs ((creaFormCuanti (snd x) f):p2)
    else let listaTerm = parseTermComas (tail $ tail $ init (snd x))
             nomPred = [head (snd x)]
         in parseForm' xs ((Pred nomPred listaTerm):p)
```

La función `parseForm` realiza una llamada a `parseForm'` con una pila vacía.

```
parseForm :: String -> Form
parseForm expr = parseForm' (infApost2 expr) []
```

4. Representación local sin nombres

Muchos lenguajes de programación, sistemas de tipos y sistemas lógicos usan variables. Diversas técnicas están disponibles para representar sintaxis con variables ligadas en un lenguaje de programación dado o en una teoría formal dada. Ahora nos enfocaremos en una representación particular de ligaduras, llamada *representación local sin nombres*[1].

Muchos problemas relacionados al ligado de variables se pueden estudiar con un lenguaje tan simple como el cálculo lambda puro. Así, primero se considerará

[1] Locally nameless representation.

la sintaxis de los términos lambda y después se hablará de la representación local sin nombres en la lógica de primer orden.

4.1. Representación con nombres

La representación más común de $\lambda-$términos recae en el uso de nombres: cada abstracción y cada variable guarda un nombre. La sintaxis de términos *crudos* con nombres se describe con la siguiente gramática.

$t := x|\lambda x.t|tt$

Y en Haskell:

```
data Term = Var String | Abs String Term | App Term Term
```

Los objetos de esta gramática se llaman "términos crudos" porque no son isomorfos a los $\lambda-$términos. Dos términos crudos `Abs ‘‘x’’ (Var ‘‘x’’)` y `Abs ‘‘y’’ (Var ‘‘y’’)` son dos objetos diferentes, aunque los dos términos lambda $\lambda x.x$ y $\lambda y.y$ deberían ser considerados iguales porque la teoría del cálculo-λ identifica a estos términos como $\alpha-$equivalentes.

4.2. Representación local con nombres

La representación local con nombres está estrechamente relacionada a la representación local con nombres. Esta representación distingue sintácticamente entre variables ligadas y libres. Las variables ligadas se representan usando un nombre (x). Las variables libres, también llamadas *parámetros*, se representan con otro tipo de nombres (p). Las abstracciones, las cuales siempre enlazan "variables ligadas", acarrean un nombre de variable ligada. La gramática de representación local con nombres se describe como sigue.

$t := x|p|\lambda x.t|tt$

Y en Haskell:

```
data Term = Bvar String | Fvar String | Abs String Term | App Term
Term
```

El interés principal de la representación local con nombres es que una variable ligada y una variable libre no se pueden confundir.

Un problema que queda en la representación local con nombres es que no son isomorfos a los $\lambda-$términos, estrictamente hablando. Dos términos pueden ser $\alpha-$equivalentes pero no ser sintácticamente iguales.

4.3. Representación de Bruijn

Usando índices de de Bruijn, uno puede construir un tipo de dato que es isomorfo al conjunto de $\lambda-$términos. En esta representación, las abstracciones no mencionan nombres y cada variable guarda un número natural que indica el número de abstracciones que debe pasar antes de alcanzar la abstracción a la cual la variable está ligada.

La gramática para términos en sintaxis de Bruijn incluyen variables construidas con un índice y abstracciones sin nombres. La i en la gramática representa un número natural.

$t := i|\lambda.t|tt$

En Haskell:

```
data Term = Var Int | Abs Term | App Term Term
```

Por ejemplo, el $\lambda-$término $\lambda x.x$ se puede representar como λ,0. Similarmente, el término $\lambda x.((\lambda y.yx)x)$ se puede representar como $\lambda.((\lambda,01)0)$.

La representación de de Bruijn sufre de una desventaja: los índices son muy sensibles a cambios en los términos en los que aparecen.

4.4. La representación local sin nombres

La representación local sin nombres combina los beneficios de la representación local con nombres con los de los índices de de Bruijn.

La gramática de la representación local sin nombres involucra un constructor para variables ligadas, construidas sobre índices de de Bruijn, y un constructor para variables libres, construidas sobre nombres. Las abstracciones no tienen nombre. La siguiente es la gramática para esta representación, donde i es un número natural y x es un nombre de variable.

$t := i|x|\lambda.t|tt$

En Haskell:

```
data Term = Bvar Int | Fvar String | Abs String Term | App Term
Term
```

Por ejemplo, el $\lambda-$término $\lambda x.xy$, el cual contiene una variable ligada x y una variable libre y, se representa como λ,0y.

4.5. Representación local sin nombres en la lógica de predicados

La última representación se puede extender a la lógica de predicados de una manera sencilla. La nueva gramática para los términos será:

$t := i|x|f(t_1, t_2, ..., t_n)$

Y para las fórmulas:

$f := \forall f|\exists f|...$

donde los puntos suspensivos indican que el resto de la gramática se queda igual. Las gramáticas en Haskell son las siguientes.

```
data TermLNR = Bvar Integer | Fvar String | FunLNR String [TermLNR]
data FormLNR = TopLNR | BotLNR | PredLNR String [TermLNR] | EqLNR
TermLNR TermLNR | NegLNR FormLNR | OrLNR FormLNR FormLNR | AndLNR FormLNR
FormLNR | ImplLNR FormLNR FormLNR | SyssLNR FormLNR FormLNR | ForallLNR
FormLNR | ExistsLNR FormLNR
```

Si se requiere transformar de representación con nombres a representación local sin nombres se debe tener un "ambiente" de las ligaduras encontradas hasta el momento. Esto es, se tendrá una lista de pares, donde cada par es un nombre de variable y el índice por el cual se va a sustituir esa variable. Por supuesto, cada vez que se encuentra un cuantificador se debe incrementar en uno el valor de cada índice.

La siguiente función toma una lista de pares de tipo `(String, Integer)` e incrementa en uno cada valor de la segunda entrada.

```
incUno :: [(String, Integer)] -> [(String, Integer)]
incUno [] = []
incUno ((var, n):xs) = (var, n+1):(incUno xs)
```

Para cambiar una variable por su índice correspondiente se necesita la función `lookup`, la cual ya viene implementada en el Prelude de Haskell. Esta función toma un elemento x y una lista de pares: devuelve Just s si el par (x, s) está en la lista o Nothing si x no está en la lista de pares. La siguiente función recibe un término, un ambiente de variables y transforma el término crudo a su equivalente en representación local sin nombre.

```
transTermLNR :: Term -> [(String, Integer)] -> TermLNR
transTermLNR (Var x) env = let indMaybe = lookup x env
    in case indMaybe of
       Just indice -> Bvar indice
       Nothing -> Fvar x
transTermLNR (Fun f terms) env = FunLNR f (map (\t -> transTermLNR t env) terms)
```

Finalmente, las siguientes funciones toman una fórmula en su representación con nombres y la transforman a su equivalente en representación local sin nombres.

```
transFormLNR' :: Form -> [(String, Integer)] -> FormLNR
transFormLNR' Top _ = TopLNR
transFormLNR' Bot _ = BotLNR
transFormLNR' (Pred p lt) env = PredLNR p (map (\t -> transTermLNR t env) lt)
transFormLNR' (Eq t1 t2) env = EqLNR (transTermLNR t1 env) (transTermLNR t1 env)
transFormLNR' (Neg f) env = NegLNR (transFormLNR' f env)
transFormLNR' (Or f1 f2) env = OrLNR (transFormLNR' f1 env) (transFormLNR' f2 env)
transFormLNR' (And f1 f2) env = AndLNR (transFormLNR' f1 env) (transFormLNR' f2 env)
transFormLNR' (Impl f1 f2) env = ImplLNR (transFormLNR' f1 env) (transFormLNR' f2 env)
transFormLNR' (Syss f1 f2) env = SyssLNR (transFormLNR' f1 env) (transFormLNR' f2 env)
transFormLNR' (Forall x f) env = let newEnv = (x, 0):(incUno env)
    in ForallLNR (transFormLNR' f newEnv)
transFormLNR' (Exists x f) env = let newEnv = (x, 0):(incUno env)
    in ExistsLNR (transFormLNR' f newEnv)

transFormLNR :: Form -> FormLNR
transFormLNR f = transFormLNR' f []
```

5. Conclusiones

Construir un analizador sintáctico de lógica de primer orden presenta una ventaja para los programas de razonamiento automatizado, ya que el usuario le dará una entrada al programa de una forma que es más entendible y más cercana al lenguaje matemático.

Por supuesto, el parser aquí presentado se puede mejorar. Una de las desventajas de este parser es que no reconoce precedencia de operadores (o conectores), así que se tienen que usar paréntesis para cada dos fórmulas. Otra desventaja es que los nombres (de constantes, variables, funciones y predicados) solo pueden tener un caracter. Un posible trabajo a futuro sería extender el parser para que reconozca cadenas de más de un caracter como nombres, y que pueda distinguir la precedencia de los conectores para no tener que usar tantos paréntesis.

La representación local sin nombres presenta una ventaja al realizar operaciones con fórmulas, pues ya no tengo que hacer verificaciones sobre $\alpha-$equivalencias. Sin embargo, transformar de representación local sin nombres a su representación con nombres no es tan sencillo, pues cada vez que veo un cuantificador tengo que introducir un nombre nuevo de variable.

Referencias

[1] Harrison, J., *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, 2009.

[2] Hutton, G., *Programming in Haskell*, Cambridge University Press, 2007.

[3] Weiss, M. A., *Data structures and algorithm analysis in Java*, Third Edition, Pearson Education, Inc., 2012.

[4] Charguéraud, A. *The Locally Nameless Representation*, *J Autom Reasoning*, (2012), 49:363–408.