Lógica Computacional

Práctica 04: Sintaxis y Semántica de la Lógica de Primer Orden

Dicter Tadeo García Rosas Erik Rangel Limón Manuel Soto Romero

Semestre 2024-2
Facultad de Ciencias UNAM

Objetivos

1. Construir un lenguaje para representar la lógica de primer orden usando los tipos de dato algebraicos de *Haskell*.
2. Definir la semántica del lenguaje que construimos para representar la lógica de primer orden.

Instrucciones

1. Para esta práctica necesitan tener instalado *GHC* y *Cabal*.

   Pueden usar una de las siguientes dos alternativas.

   - **ghcup:**
     Esta es una opción recomendada para instalar las dos herramientas usando un mismo instalador.
     Las instrucciones de cómo instalarlo para sistemas basados en *Unix* y *Windows* vienen en la página `https://www.haskell.org/ghcup/`.
     Una vez instalado, pueden verificar la instalación de `ghc` y de `cabal` utilizando el comando `ghcup tui`, el cual abrirá una interfaz de terminal (utilicen las versiones recomendadas de `ghc` y de `cabal`).
   - **Administrador de paquetes:**
     Por lo general los paquetes que necesitan llevan los nombres `ghc` y `cabal-install`, investiguen si esto difiere según su distribución.
     Debian:

     ```
     sudo apt install ghc cabal-install
     ```

     Fedora:

     ```
     sudo dnf install ghc cabal-install
     ```

2. Necesitan también la biblioteca *QuickCheck*.

   Una vez instalados *GHC* y `Cabal`, lo pueden instalar con la siguientes líneas de comandos:

   ```
   cabal update
   cabal install --lib QuickCheck
   ```

   Pueden ver si la herramienta se instaló bien si pueden interpretar con `ghci` el archivo `src/Test.hs` y no salen errores.

   En la carpeta de la práctica:

```
ghci Test.hs -package transformers
```

Pueden salir de `ghci` con `:q` o con la combinación de teclas `Ctrl+D`.

3. Resolver todas las funciones que se encuentran en el archivo `src/PrimerOrden.hs`.
4. Pueden verificar sus funciones y recibir una calificación **tentativa** con el archivo `src/Test.hs`.
   En la carpeta de la práctica:

```
runhaskell --ghc-arg='-package transformers' Test.hs
```

```
=== prop_showTerm from Test.hs:189 ===
+++ OK, passed 1000 tests.

=== prop_showFormula from Test.hs:205 ===
+++ OK, passed 1000 tests.

=== prop_libres from Test.hs:282 ===
+++ OK, passed 1000 tests.

=== prop_ligadas from Test.hs:291 ===
+++ OK, passed 1000 tests.

=== prop_subsv1 from Test.hs:340 ===
+++ OK, passed 1000 tests.

=== prop_alfaEquivalencia from Test.hs:441 ===
+++ OK, passed 1000 tests.

=== prop_unificacion from Test.hs:492 ===
+++ OK, passed 1 test.

=== prop_resolvente from Test.hs:528 ===
+++ OK, passed 1 test.

=== prop_resolucion from Test.hs:572 ===
+++ OK, passed 1 test.
```

Su calificación tentativa va a ser el número de pruebas que pasaron calificada sobre diez.

Cada ejercicio cuenta con una función `check_<nombre_de_la_funcion>`, sólo las primeras seis reciben el número de pruebas que se quieren hacer

Ejercicios

En la lógica proposicional, nuestros únicos términos eran las variables; en la lógica de primer orden, además de tener variables, también tendremos constantes, y funciones, las cuales tienen como parámetros otros términos de la lógica de primer orden.

Para denotar estos términos de la lógica de primer orden en *Haskell* crearemos el siguiente tipo de dato:

```
type Simbolo = String

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

Luego, la definición inductiva de las fórmulas de la lógica de primer orden sería la siguiente:

- Las constantes de verdad y falsedad ($\top$,$\bot$) son fórmulas de la lógica de primer orden.
- Si P es un símbolo, y $t\,s$ una lista de términos de la lógica de primer orden, entonces un predicado $P(t\,s)$ va a ser también una fórmula de la lógica de primer orden.
- Si φ y ψ son fórmulas de la lógica de primer orden, entonces $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\varphi \Leftrightarrow \psi$ son también fórmulas de la lógica de primer orden.
- Si x es un símbolo y φ una fórmula de la lógica de primer orden, entonces $\forall x.\varphi$ y $\exists x.\varphi$ también son fórmulas de la lógica de primer orden.

Esta definición inductiva será representada en *Haskell* como sigue:

```
type Simbolo = String

data Term = Var Simbolo | Fun Simbolo [Term] deriving Eq

data Formula = Verdadero | Falso
             | Predicado Simbolo [Term]
             | No Formula
             | Conj Formula Formula | Disy Formula Formula
             | Impl Formula Formula | Equiv Formula Formula
             | ForAll Simbolo Formula | Exist Simbolo Formula
             deriving Eq

infixr 4 `ForAll`, `Exist`
infixl 3 `Conj`, `Disy`
infixl 2 `Equiv`
infixr 1 `Impl`
```

También definiremos tipos sinónimo para representar las substituciones, unificadores y cláusulas; serán las siguientes:

```
type Substitucion = (Simbolo, Term)

type Unificador = [Substitucion]

type Clausula = [Formula]
```

Realiza las siguientes funciones:

1. *Instancia* `Show` *para* `Term`:

   Define una instancia `Show` para el tipo de dato `Term`. Ésta debe mostrar las variables con su respectivo nombre y a las funciones utilizando su notación tradicional.

   Hint: *Investiga cómo usar la función* `intercalate`

```haskell
instance Show Term where
  show :: Term -> String
```

   Ejemplos:

```
ghci> Var "x"
x
ghci> P.Fun "f" [Var "x", P.Fun "a" [], P.Fun "b" []]
f(x, a(), b())
```

2. *Instancia* `Show` *para* `Formula`:

   Define una instancia `Show` para el tipo de dato `Formula`.

```haskell
instance Show Formula where
  show :: Formula -> String
```

   Considera la siguiente especificación:

```
-- Constantes de verdad: V, F
-- Negación: ¬
-- Conjuncion: /\\
-- Disyuncion: \\/
-- Implicacion: ->
-- Equivalencia: <->
```

   Para denotar los cuantificadores existenciales debes utilizar los símbolos ∀ y ∃, el símbolo exacto viene en los comentarios de la función.

   Ejemplos:

```
ghci> show $ ForAll "x" (Conj (Predicado "p" [Var "x"]) (Predicado "q" [Var "y"]))
"\8704 x . (P(x) /\\ Q(y))"
ghci> show $ Exist "x" (Impl (Predicado "p" [Var "x"])
             (Predicado "q" [P.Fun "f" [Var "y", Var "z"]]))
"\8707 x . (P(x) -> Q(f(y, z)))"
```

   Nota: *Sólo es necesario utilizar el prefijo* `P.` *antes de usar el constructor* `Fun` *si interpretaron el archivo* `Test.hs`

3. *Variables libres*

   Da una función que dada una fórmula regresa la lista de todas las variables libres en ella.

```
libres :: Formula -> [String]
```

Ejemplos:

```
ghci> libres $ ForAll "x" (Predicado "P" [Var "x", Var "y", Var "z"])
["y","z"]
ghci> libres $ ForAll "x" (Exist "y" (Conj (Predicado "P" [Var "x"])
                          (Predicado "Q" [Var "y", Var "z", P.Fun "f" [Var "w"]])))
["z","w"]
```

4. *Variables ligadas*

   Da una función que dada una fórmula regresa la lista de todas las variables ligadas en ella.

```
ligadas :: Formula -> [String]
```

Ejemplos:

```
ghci> ligadas $ ForAll "x" (Predicado "P" [Var "x", Var "y", Var "z"])
["x"]
ghci> ligadas $ ForAll "x" (Exist "y" (Conj (Predicado "P" [Var "x"])
                          (Predicado "Q" [Var "y", Var "z", P.Fun "f" [Var "w"]])))
["x","y"]
```

5. *Substitución*

   Da una función que dada una fórmula y una substitución, regresa la fórmula con la substitución aplicada. Debe regresar un error si no es posible aplicar la sustitución.

```
subsv1 :: Formula -> Substitucion -> Formula
```

Ejemplos:

```
ghci> show $ subsv1 (ForAll "x" (Predicado "P" [Var "x", Var "y", Var "z"]))
            ("y", P.Fun "c" [])
"\8704 x . P(x, c(), z)"
ghci> show $ subsv1 (ForAll "x" (Predicado "P" [Var "x", Var "y", Var "z"]))
            ("y", P.Fun "c" [Var "x"])
"*** Exception: No es posible sustituir variables ligadas
CallStack (from HasCallStack):
  error, called at ./PrimerOrden.hs:114:31 in main:PrimerOrden
```

6. *α-equivalencia*

   Da una función que determine si dos fórmulas son *alfa-equivalentes*.

```
alfaEquivalencia :: Formula -> Formula -> Bool
```

Ejemplos:

```
ghci> alfaEquivalencia (ForAll "x" (Predicado "P" [Var "x", Var "y", Var "z"]))
      (ForAll "w" (Predicado "P" [Var "w", Var "y", Var "z"]))
True
ghci> alfaEquivalencia (ForAll "x" (Predicado "P" [Var "x", Var "y", Var "z"]))
      (ForAll "w" (Predicado "P" [Var "w", Var "y", P.Fun "a" []]))
False
```

7. *Unificación:*

   Dadas dos fórmulas de la lógica de primer orden, regresa el unificador más general de ambas, si es que existe.

```
unifica :: Formula -> Formula -> Maybe Unificador
```

El algoritmo de Martelli-Montanari recibe un conjunto de ecuaciones W de la forma $\{s_1 = r_1, \ldots, s_n = r_n\}$, siendo s_i y r_i fórmulas de la lógica de primer orden. Inicialmente, si queremos unificar dos fórmulas φ y ψ, comenzaremos con $W = \{\varphi = \psi\}$. El algoritmo modificará el unificador σ recursivamente a partir de revisar cada ecuación en W siguiendo las siguientes reglas:

1. $\boxed{X = X}$
   Eliminamos la ecuación, es decir, continuamos el algoritmo con $W \setminus \{X = X\}$.
2. $\boxed{X = t \text{ ó } t = X}$
   Si $X \in vars(t)$*, falla; en otro caso, en* σ *aplicamos a todos lo términos la sustitución* $[X := t]$ *y le añadimos la sustitución en cuestión, y por último, continuamos con el conjunto* $(W \setminus \{X = t\})_{[X:=t]}$*, es decir, quitamos la ecuación* $X = t$ *o* $t = X$ *según sea el caso y le aplicamos la sustitución a cada ecuación.*
3. $\boxed{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$
   Continuamos el algoritmo con $\{s_1 = t_1, \ldots, s_n = t_n\} \cup (W \setminus \{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\})$.
4. $\boxed{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n)}$
   Falla.
5. $\boxed{P(s_1, \ldots, s_n) = P(t_1, \ldots, t_n)}$
   Continuamos el algoritmo con $\{s_1 = t_1, \ldots, s_n = t_n\} \cup (W \setminus \{P(s_1, \ldots, s_n) = P(t_1, \ldots, t_n)\})$.
6. $\boxed{P(s_1, \ldots, s_n) = Q(t_1, \ldots, t_n)}$
   Falla.
7. $\boxed{\varphi \oplus \psi = \varphi' \oplus \psi'}$ Siendo $\oplus$ un operador binario.
   Continuamos el algoritmo con $\{\varphi = \varphi', \psi = \psi'\} \cup (W \setminus \{\varphi \oplus \psi = \varphi' \oplus \psi'\})$

En los casos no mencionados, falla.

Ejemplos:

```
ghci> unifica (Predicado "Q" [Var "x", Var "y", Var "x"])
              (Predicado "Q" [Var "y", Fun "g" [Var "x"], Var "x"]).
Nothing
ghci> unifica (Predicado "Q" [Fun "a" [], Fun "c" [], Fun "f" [Var "x"]])
              (Predicado "Q" [Var "z",Vay "y", Var "v"])
Just [("z", Fun "a" []), ("y", Fun "c" []), ("v", Fun "f" [Var "x"])]
```

8. *¿Existe una resolvente?*

   La regla de inferencia de la resolución binaria para la lógica de primer orden es la siguiente:

$$\frac{\varphi \vee \alpha \quad \psi \vee \neg\beta \quad Var(\varphi) \cap Var(\psi) = \emptyset \quad \sigma = umg(\alpha, \beta)}{(\varphi \vee \psi)_\sigma}$$

   Siendo φ y ψ cláusulas, y α y β literales.

   Da una función que verifique que se cumplen estas condicione; que haya dos literales (a saber α y β) que ambas sean unificables y en las cláusulas sean complementarias una de otra (que una esté negada y la otra no), y que la interseccion de variables del resto de literales de cada cláusula sea vacía.

```
resolvente :: Clausula -> Clausula -> Bool
```

   Ejemplos:

```
ghci> resolvente [Predicado "P" [Var "x"], Predicado "Q" [P.Fun "f" [Var "x"]]]
      [Predicado "P" [Var "z"], No (Predicado "Q" [P.Fun "f" [Var "y"]])]
True
ghci> resolvente [Predicado "P" [Var "x"], Predicado "Q" [P.Fun "f" [Var "x"]]]
      [Predicado "P" [Var "x"], No (Predicado "Q" [P.Fun "f" [Var "y"]])]
False
```

9. *Calcula la resolvente*

   Da una función que calcula la resolvente de dos fórmulas usando la regla de inferencia anetrior.

   Notas importantes:

   - Debes asumir que siempre se puede obtener una resolvente, por ejemplo si una literal en una cláusula no es candidata para hacer la resolución, ésta entonces debe aparecer en la cláusula final.
   - Debes calcular únicamente la resolvente con la primer literal que sea candidata de la primer cláusula.

```
resolucion :: Clausula -> Clausula -> Clausula
```

   Ejemplos:

```
ghci> resolucion [Predicado "P" [Var "x"], Predicado "Q" [P.Fun "f" [Var "x"]]]
      [Predicado "P" [Var "z"], No (Predicado "Q" [P.Fun "f" [Var "y"]])]
[P(y),P(z)]
ghci> resolucion [Predicado "P" [Var "x"]]
      [No (Predicado "P" [P.Fun "f" [Var "a", Var "b", Var "c"]])]
[]
ghci> resolucion [Predicado "P" [Var "x",Var "y"], Predicado "R" [Var "x",Var "y"]]
      [No (Predicado "P" [P.Fun "f" [Var "y",Var "z"], P.Fun "c" []]),
       Predicado "R" [Var "w",P.Fun "a" []]]
[R(f(c(), z), c()),R(w, a())]
```

Entrega

1. La tarea se entrega en *Github Classroom*.
2. La tarea se entrega en equipos de hasta 4 personas.
3. Además de resolver los ejercicios, deben entregar un `README` (`.md` o `.org`) en la raiz de la carpeta de la práctica que contenga los datos de todos los integrantes, y por cada una de sus funciones dejar una breve explicación de su solución.

Consideraciones

1. Todas las prácticas con copias totales o parciales tanto en el código como en el `README` serán evaluadas con cero.
2. Las únicas funciones con soluciones iguales admisibles son todas aquellas que sean iguales a las resueltas por el grupo en el laboratorio, sin embargo la explicación de su solución en el `README` debe ser única para cada equipo.
3. No entregar el `README` o tenerlo incompleto se penalizará con hasta dos puntos menos sobre su calificación en la práctica.
4. Cada día de retraso se penalizará con un punto sobre la calificación de la práctica.
5. Pueden usar las funciones `map, filter, reverse, sum, elem, all, any, head, tail`. Pueden utilizar otras funciones siempre y cuando no resuelvan directamente el ejercicio, expliquen en el `README` qué es lo que hace, y pertenezca a alguna biblioteca estándar de *Haskell* (es decir, no es válido utilizar paqueterías que requieran una instalación manual).
6. Los tests de ésta práctica tardan entre uno y tres minutos. Al final del archivo `src/PrimerOrden.hs` hay una variable `pruebas`, ésta indica el número de pruebas que se realizarán a cada una de las funciones. Por defecto está configurada para hacer 1000 pruebas, pero pueden ponerle un valor menor si consideran que las pruebas tardan mucho en completarse, siempre y cuando éste valor no sea menor a 100. Si las pruebas tardan más de los tres minutos aún con 100 pruebas, consideren otra solución.
7. Si a la variable `pruebas` le pusieron un valor menor a 1000, asegúrense que pasen las pruebas de manera consistente, es decir, que puedan ejecutar varias veces las pruebas sin dar errores.
8. Pueden hacer tantas funciones auxiliares como quieran, pero no deben modificar la firma de las funciones ni de las variables, ni la definición de tipos de dato que se les dió.
9. No se recibirán prácticas que no compilen (no debe arrojar errores la orden `ghci Test.hs`). Si no resuelven alguna de las funciones déjenlas como `undefined`, pero no eliminen la función, ya que ésto lanzará errores.
10. No deben modificar el archivo `src/Test.hs`. Si encuentran errores o tienen dudas sobre las pruebas, manden un correo al ayudante de laboratorio (si encuentran errores tendrán una participación).
11. Las participaciones en el laboratorio se aplicarán de manera individual sobre la calificación que tuvieron en la práctica.