Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI2661 - Laboratorio de Lenguajes I
Enero-Abril 2009

Primer Proyecto:
Programación lógica en Haskell

1 Programación lógica

Definiciones preliminares. Asumimos inicialmente la existencia de cuatro alfabetos distintos que forman el **conjunto de símbolos no-lógicos**:

–> Constantes: $\{a, b, c, ...\}$
–> Variables: $\{X, Y, Z, ...\}$
–> Funciones: $\{f, g, h, ...\}$
–> Predicados: $\{p, q, r, ...\}$

Además de un conjunto de **símbolos lógicos**:

–> Conectores: $\{\neg, \wedge, \vee, \Rightarrow, \Leftarrow, \Leftrightarrow\}$.
–> Cuantificadores: $\{\forall, \exists\}$.

Cada función y cada predicado tiene asociado un número finito n de argumentos o n-aridad.

Presentamos una definición de la lógica de predicados usando el tipo árbol.

Un **término** es un árbol T que satisface las siguientes condiciones:

i) las hojas de T son variables o constantes,

ii) todo nodo no-terminal de T es un término de la forma $f(\tau_1, ..., \tau_n)$ y tiene los términos $\tau_1, ..., \tau_n$ como sus n sucesores inmediatos.

Si $\tau_1, ..., \tau_n$ son símbolos que designan términos y p es un predicado, entonces la **fórmula atómica** $p(\tau_1, ..., \tau_n)$ es un árbol y sus n sucesores son los subárboles $\tau_1, ..., \tau_n$.

Una **fórmula bien formada** (o simplemente **fórmula**) es un árbol tal que:

1) las hojas son fórmulas atómicas,

2) un nodo no-terminal con una única fórmula sucesora ϕ es alguno entre

$\neg\phi$	negación
$\forall X[\phi(X)]$	cuantificación universal
$\exists X[\phi(X)]$	cuantificación existencial

3) un nodo no-terminal con dos fórmulas sucesoresa ϕ_1 y ϕ_2, es alguno entre

$\phi_1 \wedge \phi_2$	conjunción
$\phi_1 \vee \phi_2$	disyunción
$\phi_1 \Rightarrow \phi_2$	implicación
$\phi_1 \Leftarrow \phi_2$	condicional
$\phi_1 \Longleftrightarrow \phi_2$	equivalencia

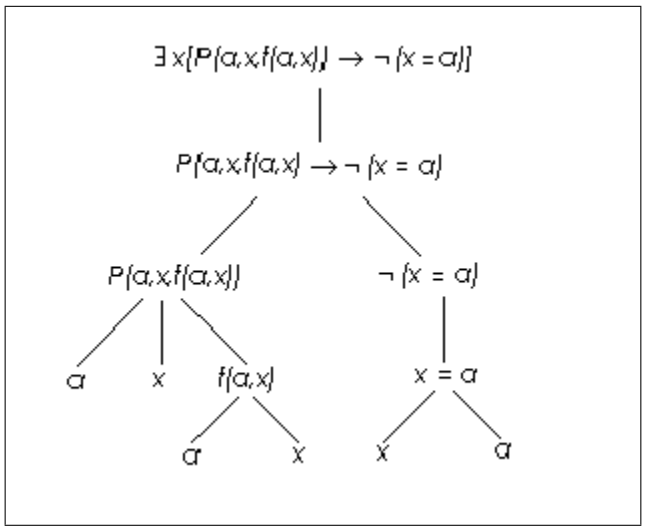


Figure 1:

La figura muestra un ejemplo del árbol que representa la fórmula

$$\exists X[p(a, X, f(a, X)) \rightarrow \neg = (X, a)]$$

La **forma normal de cláusulas** es el sublenguaje del lenguaje de la lógica de predicados en el cual se expresa la programación lógica convencional, y el lenguaje PROLOG en particular.

Existe un algoritmo ([2] y [1]) que convierte cualquier fórmula en la forma normal, usando un procedimiento de reducción semántica (llamado "escolemización" en honor al lógico noruego Th. A. Skolem) que permite eliminar todos los cuantificadores existenciales: omitiendo la presencia de los cuantificadores universales, la fórmula inicial es transformada en una conjunción de disyunciones

$$\begin{aligned}
&\left(\phi_{1_1}\left(\bar{t}_{1_1}\right) \vee ... \vee \phi_{1_{i_1}}\left(\bar{t}_{1_{i_i}}\right) \vee \neg\phi_{1_{i_1+1}}\left(\bar{t}_{1_{i_i+1}}\right) \vee ... \vee \neg\phi_{1_{t_1}}\left(\bar{t}_{1_{k_1}}\right)\right) \wedge \\
&\vdots \\
&\wedge \left(\phi_{n_1}\left(\bar{t}_{n_1}\right) \vee ... \vee \phi_{n_{i_n}}\left(\bar{t}_{n_{i_n}}\right) \vee \neg\phi_{n_{i_n+1}}\left(\bar{t}_{1_{i_n+1}}\right) \vee ... \vee \neg\phi_{n_{t_n}}\left(\bar{t}_{n_{k_n}}\right)\right)
\end{aligned}$$

donde $\bar{t}_{j_r}$ es una lista de términos para todo $1 \leq j \leq n$ y $1 \leq r \leq n_{k_n}$.

Se escribe la expresión línea por línea, sin referencia explícita a las conjunciones, donde cada línea es un grupo disyuntivo aparte llamado **cláusula**

$$\begin{aligned}
&\phi_{1_1}\left(\bar{t}_{1_1}\right) \vee ... \vee \phi_{1_{i_1}}\left(\bar{t}_{1_{i_i}}\right) \vee \neg\phi_{1_{i_1+1}}\left(\bar{t}_{1_{i_1+1}}\right) \vee ... \vee \neg\phi_{1_{k_1}}\left(\bar{t}_{1_{k_1}}\right) \\
&\vdots \\
&\phi_{n_1}\left(\bar{t}_{n_1}\right) \vee ... \vee \phi_{n_{i_n}}\left(\bar{t}_{n_{i_n}}\right) \vee \neg\phi_{n_{i_n+1}}\left(\bar{t}_{1_{i_n+1}}\right) \vee ... \vee \neg\phi_{n_{k_n}}\left(\bar{t}_{n_{k_n}}\right)
\end{aligned}$$

Luego, cada cláusula , por las leyes de implicación y de De Morgan, adquiere la forma

$$\phi_{j_1}(\bar{t}_{j_1}) \vee ... \vee \phi_{j_{i_j}}\left(\bar{t}_{j_{i_j}}\right) \Longleftarrow \phi_{j_{i_j+1}}\left(\bar{t}_{j_{i_j+1}}\right) \wedge ... \wedge \phi_{j_{k_j}}\left(\bar{t}_{j_{k_j}}\right)$$

para todo $1 \leq j \leq n$.

Una **cláusula de Horn** es una cláusula donde a lo sumo una fórmula atómica es positiva. Las cláusulas de Horn se clasifican en:

1) **Reglas**; cuando una fórmula atómica es positiva y al menos una fórmula atómica es negativa

$$\phi_{j_1}(\bar{t}_{j_1}) \Longleftarrow \phi_{j_2}(\bar{t}_{j_2}) \wedge ... \wedge \phi_{j_{k_j}}\left(\bar{t}_{j_{k_j}}\right)$$

a $\phi_{j_1}(\bar{t}_{j_1})$ se le llama **cabeza** y a la conjunción $\phi_{j_2}(\bar{t}_{j_2}) \wedge ... \wedge \phi_{j_{k_j}}\left(\bar{t}_{j_{k_j}}\right)$ **cuerpo**.

2) **Hechos**; fórmulas atómicas positivas $\phi_{j_1}(\bar{t}_{j_1})$, sin cuerpo.

3) **Metas**, fórmulas atómicas negativas

$$\neg\phi_{j_1}(\bar{t}_{j_1}) \vee ... \vee \neg\phi_{j_{k_j}}\left(\bar{t}_{j_{k_j}}\right)$$

que se convierte en la expresión

$$false \Leftarrow \phi_{j_1}(\bar{t}_{j_1}) \wedge ... \wedge \phi_{j_{k_j}}\left(\bar{t}_{j_{k_j}}\right)$$

Una **cláusula vacía** no tiene fórmulas atómicas y por definición no satisface ninguna interpretación, es siempre $false$.

Las reglas y los hechos, que son fórmulas atómicas positivas con o sin cuerpo, son **cláusulas definidas**. Una fórmula lógica expresada como una conjunción de cláusulas definidas se llama **programa lógico**. Una fórmula lógica expresada como una cláusulas de Horn del tipo meta se llama **pregunta**.

El lenguaje (muy reducido) de los programas lógicos y las preguntas se genera con una gramática en forma de Backus-Naur extendida del tipo:

$$\begin{aligned}
Prolog ::= &\quad \{AtomForm\ '\Leftarrow'\ Tail\ '.' \mid AtomForm\ '.'\}^0 \\
AtomForm ::= &\quad \{a-z\}^1\ ['('ListaTerm')'] \\
ListaTerm ::= &\quad Term\,\{','\ Term\}^0 \\
Term ::= &\quad Const \mid Var \mid Fun \\
Fun ::= &\quad \{a-z\}^1\ '('\ ListaTerm\ ')' \\
Var ::= &\quad \{A-Z\}^1 \\
Const ::= &\quad \{a-z\}^1 \mid \{1-9\}^1\,\{0-9\}^0 \\
Tail ::= &\quad AtomForm\,\{','\ AtomForm\}^0 \\
Preg ::= &\quad '\Leftarrow'\ Tail
\end{aligned}$$

El tipo de datos en Haskell para un programa lógico a partir de esta gramática simplificada podría ser:

$$
\begin{array}{lll}
\text{data} & Value = & I\ Int \\
 & & \mid B\ Bool \\
 & & \mid F\ Float \\
 & & \mid Ch\ Char \\
 & & \mid St\ String \\
\text{data} & Term = & Const\ String \\
 & & \mid Var\ String \\
 & & \mid Fun\ String\ [Term] \\
\text{data} & Atom = & (String, [Term]) \\
\text{type} & Tail = & [Atom] \\
\text{data} & Definite = & Fact\ Atom \\
 & & \mid Rule\ Atom\ Tail \\
\text{data} & Program = Prolog\ [Definite] & \\
\text{Newtype} & Question = & Q\ Tail \\
\text{type} & Subs = & [(Term, Term)]
\end{array}
$$

2 Regla de inferencia lógica SLD-*Resolución*

Una **sustitución** es un conjunto de remplazos de variables por términos en una fórmula atómica

$$\sigma = \{X_{i_1} \leftarrow t_{i_1}, \ldots, X_{i_k} \leftarrow t_{i_k}\}$$

donde $X_1, \ldots, X_n$ son las variables que ocurren en la fórmula. Nótese que una sustitución no afecta necesariamente todas las variables de la fórmula. La fórmula atómica $\sigma(\phi)$ que resulta de una sustitución σ se llama **instancia** de la fórmula original ϕ.

Las sustituciones deben cumplir dos condiciones:

1) Funcionalidad: $X_{i_u} \neq X_{i_v}$ para todo $1 \leq u, v \leq k$ con $u \neq v$.

2) Idempotencia: $\sigma(\sigma(\phi)) = \sigma(\phi)$.

Dada la fórmula ϕ y las sustituciones

$$
\begin{aligned}
\sigma_1 &= \{X_{i_1} \leftarrow t_{i_1}, \ldots, X_{i_k} \leftarrow t_{i_k}\} \\
\sigma_2 &= \{Y_{i_1} \leftarrow s_{i_1}, \ldots, Y_{i_m} \leftarrow s_{i_m}\}
\end{aligned}
$$

se define su **composición** como la sustitución

$$\sigma_2 \circ \sigma_1 = \{\sigma_2(X_{i_1} \leftarrow t_{i_1}), \ldots, \sigma_2(X_{i_k} \leftarrow t_{i_k})\} \cup \{Y_{i_j} \leftarrow s_{i_j} \mid Y_{i_j} \notin \{X_{i_1}, \ldots, X_{i_k}\}\}$$

que se interpreta de la siguiente manera: la aplicación de σ_1 a ϕ introduce, eventualmente, nuevas variables que provienen de los términos $t_{i_1}, \ldots, t_{i_k}$, luego, la aplicación de σ_2 a $\sigma_1(\phi)$ debe darse tanto en las nuevas variables como en aquellas otras que no afecta la primera sustitución.

Un **unificador** para las fórmulas ϕ_1 y ϕ_2 es una sustitución σ, definida sobre la unión de las variables de ambas, tal que

$$\sigma(\phi_1) = \sigma(\phi_2)$$

Un unificador μ se dice **general** cuando, para cualquier unificador σ existe una sustitución ρ tal que

$$\sigma = \rho \circ \mu$$

Un unificador general es la mínima sustitución que iguala las instancias de ambas fórmulas y permite, eventualmente, otras sustituciones posteriores que remplacen las variables restantes.

La regla de inferencia SLD**-resolución** usa como premisas una pregunta y una regla del programa lógico: Se instancian ambas fórmulas (la pregunta y la regla) con el unificador general de la primera fórmula atómica de la pregunta y la cabeza de la regla, que por supuesto, deben compartir el mismo predicado y tener la misma aridad.

Formalmente, sean la pregunta (llamada en este contexto **padre central**)

$$\neg\varphi_1\left(t_{1_1},,t_{1_{k_1}}\right)\vee\neg\varphi_2\left(t_{2_1},,t_{2_{k_2}}\right)\vee\cdots\vee\neg\varphi_m\left(t_{m_1},,t_{m_{k_m}}\right)$$

y el programa lógico cuya primera cláusula definida (de allí la D, por $definida$) en el orden secuencial de las cláusulas (llamada en este contexto **padre lateral**) es,

$$\phi_1\left(s_{1_1},,s_{1_{j_1}}\right)\Leftarrow\phi_2\left(s_{2_1},,s_{2_{j_2}}\right)\wedge\cdots\wedge\phi_n\left(s_{n_1},,s_{n_{j_n}}\right)$$

cumpliendo las condiciones;

i) el predicado de la primera fórmula atómica de la pregunta (de allí la S, por $selectiva$) es el mismo predicado de la cabeza de la cláusula definida, es decir, $\varphi_1 = \phi_1$, y

ii) la aridad del predicado φ_1 es la misma que la del predicado ϕ_1 es decir, $k_1 = j_1$,

Se construye entonces la conclusión, llamada **resolvente**, como otra pregunta

$$\neg\phi_2\left(s_{2_1},,s_{2_{j_2}}\right)\vee\cdots\vee\neg\phi_n\left(s_{n_1},,s_{n_{j_n}}\right)\vee\neg\varphi_2\left(t_{2_1},,t_{2_{k_2}}\right)\vee\cdots\vee\neg\varphi_m\left(t_{m_1},,t_{m_{k_m}}\right)$$

que, de nuevo, será sometida al programa (de allí la L , por $lineal$).

Si se busca una respuesta afirmativa cuando se somete la pregunta φ al programa Φ, el proceso sistemático del empleo de la regla SLD-resolución debe culminar con una última unficación cuyo resolvente es la meta cláusula vacía $false$ (recuérdese que $false$ es el elementro neutro del conector lógico binario $\vee$), es decir, el sistema de inferencia despliega una prueba por contradicción, por ello tiene éxito cuando se concluye $false$. Si ninguna instancia de la pregunta φ puede ser derivada lógicamente de Φ, por medio de SLD-resolución, entonces se produce la respuesta No. En el lenguaje de la teoría de pruebas esto es

$$\Phi\wedge\neg\varphi\nvdash false$$

3 Algoritmo de Robinson

Dadas dos fórmulas atómicas:

$$\phi_1\left(t_1,\ldots,t_n\right)\text{ y }\phi_2\left(s_1,\ldots,s_m\right)$$

donde $t_1, \ldots, t_n, s_1, \ldots, s_m$ son términos (*i.e.* constantes, variables o funciones), el algoritmo de Robinson permite saber si ambas fórmulas atómicas son unificables, en cuyo caso devuelve el unificador general. En caso contrario, devuelve un mensaje de error razonando la falla. El algoritmo se implementa con dos pilas; la primera, Υ, para guardar los pares de términos (t_i, s_i) según aparecen, componente a componente, en las tuplas que forman los argumentos de los predicados comparados, es decir, (t_1, s_1) en el tope de la pila y (t_n, s_n) en el fondo; la segunda, Φ, para guardar las sutituciones sucesivas cuya composición constituye el unificador general, una entrada por variable a remplazar. Obviamente, el algoritmo falla si $\phi_1 \neq \phi_2$ o $n \neq m$.

Para facilitar la comprensión del algoritmo, categorizamos el álgebra de los términos T en tres clases; C para las constantes, V para las variables, F para las funciones. A continuación describimos informalmente el algoritmo:

.

Algoritmo de Robinson

(in $(\phi_1(t_1, \ldots, t_n), \phi_2(s_1, \ldots, s_m))$ / out $\mu :: \{V \leftarrow T\}$)

if $\phi_1 \neq \phi_2$ then *mensaje de error*			
else if $n \neq m$ then *mensaje de error*			
else while $\Upsilon \neq \varnothing$ do			
$pop_{\Upsilon}\ (t_i, s_i)$			
$(t, s) \leftarrow instancia\,(t_i, s_i)$			
if	$(t, s) \in C \times C \wedge t \neq s$	then	*mensaje de error*
else if	$(t, s) = (f(\ldots), g(\ldots)) \in F \times F \wedge f \neq g$	then	*mensaje de error*
else if	$(t, s) = (f(\ldots), s) \in F \times C \vee$ $(t, s) = (t, g(\ldots)) \in C \times F$	then	*mensaje de error*
else if	$(t, s) = \left(f\left(u_1, \ldots, u_{n_f}\right), g\left(v_1, \ldots, v_{n_g}\right)\right) \in F \times F$ $\wedge f = g \wedge n_f = n_g$	then	$push_{\Upsilon}\ (u_1, v_1),$ $\ldots, push_{\Upsilon}\left(u_{n_f}, v_{n_g}\right)$
else if	$(t, s) \in V \times V$	then	$push_{\Phi}\ (t \leftarrow s)$
else if	$(t, s) = (t, g(\ldots)) \in V \times F \wedge$ $occurs\text{-}check\,(t, s) = true$	then	*mensaje de error*
else if	$(t, s) = (f(\ldots), s) \in F \times V$ $occurs\text{-}check\,(t, s) = true$	then	*mensaje de error*
else if	$(t, s) \in V \times C \cup V \times F$	then	$push_{\Phi}\ (t \leftarrow s)$
else if	$(t, s) \in C \times V \cup F \times V$	then	$push_{\Phi}\ (s \leftarrow t)$
end while			
$\mu \leftarrow idemp\,(\Phi)$			

.

El algoritmo llama a tres procedimientos;

1) $instancia\,(t_i, s_i)$ es la función que aplica al par (t_i, s_i), en cada ciclo, las sustituciones que se van acumulando en Φ y devuelve el par actualizado (t, s).

2) $occurs\text{-}check\,(t, s)$ es la función booleana que busca dentro del término t la ocurrencia de la variable s, o viceversa.

3) $idemp\,(\Phi)$ garantiza que μ sea idempotente calculando el mínimo n para

el cual

$$\mu^n = \mu^{n+1}$$

por lo cual μ^n será el único unificador general.

4 Backtracking

Todas las cláusulas definidas que comparten el predicado de la fórmula atómica positiva constituyen una **definición** (o **procedimiento**). La regla de inferencia SLD-resolución busca las soluciones (*i.e.* la unificación general) que se derivan de la unificación de la primera fórmula atómica con la primera cláusula definida en la definición que se corresponde con el predicado. El proceso de unificar secuencialmente todas las cláusulas de una definición se llama **backtracking**. Posiblemente, algunas de las cláusulas produzcan soluciones positivas (unificaciones generales) y otras no lo hagan. La estructura que describe este proceso de búsqueda de soluciones se llama SLD-**árbol**.

5 Ejemplo

En Prolog, como en Haskell, la estrutura de datos básica es la lista, y se entiende como una función algebraica

$$\begin{aligned} dispar\ ([], list): &\quad 1 \mid a \times (Lista\ a) \rightarrow Lista\ \ a \\ dispar\ ([], list)\ (*) = &\quad [] \\ dispar\ ([], list)\ (head, tail) = &\quad list\,(head, tail) \end{aligned}$$

Sea el programa lógico

$$\begin{aligned} &\text{c1:} \quad append\,([], X, X)\,. \\ &\text{c2:} \quad append\,(list\,(A, X)\,, Y, list\,(A, Z)) \Leftarrow append\,(X, Y, Z)\,. \end{aligned}$$

y la pregunta

$$?append\,(list\,(Head, Tail1)\,, Tail2, list\,(Head, list\,(a, [])))$$

El sistema de inferencia SLD-resolución trata primero de resolver la pregunta con la cláusula c1 pero fracasa porque no puede unificar el par de términos $([], list\,(H, T1))$. Intenta luego unificar las fórmulas atómicas

$$\begin{aligned} &append\,(list\,(Head, Tail1)\,, Tail2, list\,(Head, list\,(a, [])))\\ &append\,(list\,(A, X)\,, Y, list\,(A, Z)) \end{aligned}$$

devolviendo el unificador general

$$\mu_1 = \{A \leftarrow Head, X \leftarrow Tail1, Y \leftarrow Tail2, Z \leftarrow list\,(a, [])\}$$

El proceso de la inferencia lógica es el siguiente:

padre central:	$\neg append\left(list\left(Head, Tail1\right), Tail2, list\left(Head, list\left(a, []\right)\right)\right)$
padre lateral:	$append\left(list\left(Head, Tail1\right), Tail2, list\left(Head, list\left(a, []\right)\right)\right) \vee \neg append\left(Tail1, Tail2, list\left(a,\right.\right.$
resolvente:	$\neg append\left(Tail1, Tail2, list\left(a, []\right)\right)$

con lo cual el sistema debe ahora resolver la pregunta

$$?append\left(Tail1, Tail2, list\left(a, []\right)\right).$$

unificando las fórmulas atómicas

$$append\left(Tail1, Tail2, list\left(a, []\right)\right)$$
$$append\left([], X, X\right)$$

con el unificador general

$$\mu_2 = \{Tail1 \leftarrow [], Tail2 \leftarrow list\left(a, []\right)\}$$

en el proceso de refutación

padre central:	$\neg append\left([], list\left(a, []\right), list\left(a, []\right)\right)$
padre lateral:	$append\left([], list\left(a, []\right), list\left(a, []\right)\right)$
resolvente:	$false$

Si se aplica backtracking, el sistema busca otra solución a partir de la última elección de padre lateral, es decir, intentará unificar ahora las fórmulas atómicas

$$append\left(Tail1, Tail2, list\left(a, []\right)\right)$$
$$append\left(list\left(A, X\right), Y, list\left(A, Z\right)\right)$$

consiguiendo el unificador general

$$\mu_3 = \{Tail1 \leftarrow list\left(A, X\right), Y \leftarrow Tail2, A \leftarrow a, Z \leftarrow []\}$$

en el proceso de inferencia

padre central:	$\neg append\left(list\left(a, X\right), Tail2, list\left(a, []\right)\right)$
padre lateral:	$append\left(list\left(a, X\right), Tail2, list\left(a, []\right)\right) \vee \neg append\left(X, Tail2, []\right)$
resolvente:	$\neg append\left(X, Tail2, []\right)$

Finalmente, la pregunta

$$?append\left(X, Tail2, []\right)$$

con el unificador general

$$\mu_4 = \{X \leftarrow [], Tail2 \leftarrow []\}$$

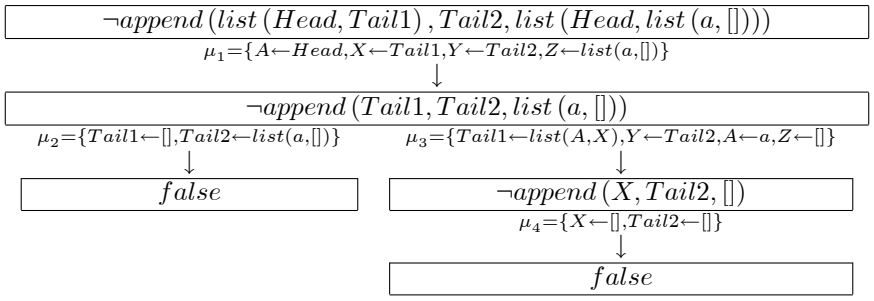
se resuelve en el proceso de refutación

padre central:	$\neg append\left([], [], []\right)$
padre lateral:	$append\left([], [], []\right)$
resolvente:	$false$

Las soluciones a la consulta son, primero

$Head =?$

$Tail1 = []$
$Tail2 = list\,(a, [])$
Luego del backtracking,
$Head =?$
$Tail1 = list\,(a, [])$
$Tail2 = []$
Todo el proceso de inferencia se suele representar en el SLD-árbol



Observación: Para evitar la repetición en los nombres de las variables entre el padre central y el padre lateral, el interpretador debe renombrar las variables de la cláusula del programa lógico con nombres de variables que se sepa no estan en uso.

En la estructura de tipos de datos sugerida antes, tanto el programa lógico como la pregunta se expresan así:

```
app :: Program
app =  Prolog [  Fact  ("append",[Fun "list" [ ], Var "X", Var "X"]),
                 Rule  ("append",[Fun "list" [Var "A",Var "X"], Var "Y", Fun "list" [Var "A",Var "
                       [("append",[Var"X",Var"Y",Var"Z"])]
               ]
y
query :: Question
query =      Q [  ("append",  [Fun "list" [Var "H",Var "T1"],
                               Var "T2",
                               Fun "list" [Var "H",Fun "list" [Const (Ch 'a'),Fun "list" [ ]]
                               ]
                  )
               ]
              ]
```

6 Enunciado del proyecto

- Se debe programar en haskell las siguientes funciones

$$\begin{array}{rcl} \text{substitucion} & : & : Atom \to Subs \to Atom \\ \text{robinson} & : & : Atom \to Atom \to Subs \\ \text{resolucion} & : & : Program \to Question \to Subs \\ \text{backtracking} & : & : Program \to Question \to [Subs] \end{array}$$

  donde *Subs* es el tipo de datos que implementa los unificadores generales

$$\text{type} \quad Subs = \quad [(Term, Term)]$$

- Su programa deben manejar correctamente los mensajes de error cuando no sea posible unificar dos fórmulas atómicas, o cuando SLD-resolución no pueda llegar a la cláusula vacía.

- Su programa debe incorporar funciones de *pretty-print* para presentar las fórmulas, los términos, las sustituciones y todas las soluciones del backtracking de manera legible.

Documentación a entregar:

- El módulo *Prologs.hs* conteniendo el código fuente Haskell que implementa las funciones que se piden, correctamente documentado utilizando la herramienta *Haddock*.

- Un $Makefile$ que permita compilar el programa y generar la documentación. El programa principal será compilado y utilizado desde la línea de comandos.

- **Fecha de Entrega.** Viernes 20 de Febrero de 2009 (Semana 6).

- **Valor de Evaluación.** Treinta (30) puntos.

References

[1] Lloyd, J. W. (1984)
Foundations of Logic Programming
`Springer-Verlag`

[2] Hogger, C. J. (1990)
Essentials of Logic Programming
`Oxford University Press`