Aprendizaje automático de programas lógico-funcionales

Cèsar Ferri Ramírez, José Hernández Orallo, María José Ramírez Quintana

Departamento de Sistemas Informáticos y Computación (DSIC)
Universidad Politécnica de Valencia
Camino de Vera s/n, Valencia (Spain)
{cferri,jorallo,mramirez}@dsic.upv.es

Resumen

En este trabajo se presenta un sistema para el aprendizaje de programas lógico-funcionales a partir de ejemplos y de conocimiento previo. Esto supone una extensión de la programación lógica inductiva. El marco se basa en dos operadores fundamentales, la generalización consistente restrictiva, inspirado en los sistemas de aprendizaje basados en generalización, y el narrowing inverso, inspirado en la inversión del operador deductivo del lenguaje de representación, en este caso, los lenguajes lógico-funcionales. Dichos operadores se combinan en un algoritmo con un carácter marcadamente evolutivo en el que dos niveles de poblaciones (de reglas y de programas sobre éstas) se van modificando y combinando hasta llegar a una solución satisfactoria según el criterio de selección de hipótesis. El resultado es el sistema FLIP, un sistema de aprendizaje sobre un lenguaje de representación universal de fácil inteligibilidad que permite, entre otros, la inducción con o sin conocimiento previo de programas recursivos, de árboles de decisión, y de clasificadores sobre datos no estructurados.

Palabras Clave: Aprendizaje Automático, Programación Lógico-Funcional, Programación Lógica Inductiva (ILP).

1 Introducción

La programación lógica inductiva (ILP) [9] es un marco para la inferencia inductiva de teorías de primer orden a partir de hechos. El uso de la programación lógica para el aprendizaje se justifica en el hecho de que los programas lógicos son una representación simple de los ejemplos, el conocimiento previo (*background knowledge*) y las hipótesis. El aprendizaje ILP puede ser considerado como el proceso de inferencia de una teoría P (un programa lógico) desde hechos (en general, evidencias positiva y negativa). La ILP ha significado un importante salto cualitativo en el campo del aprendizaje automático ya que posee una expresividad mayor que los tradicionales marcos proposicionales.

Sin embargo, los lenguajes de programación lógicos como el Prolog (el más extendido de este paradigma) carecen de varias características de programación útiles tales como funciones evaluables, tipos, orden superior y evaluación perezosa. Aunque estas características las proporciona la programación funcional, ésta no posee otras típicas de la programación lógica como las variables lógicas y la unificación. Por ello, en los últimos años ha crecido el interés por la *integración* de ambos paradigmas [4].

Los lenguajes integrados (véase, e.g. Curry [5]) explotan plenamente las ventajas de la programación declarativa en sentido general: funciones, predicados e igualdad. Los lenguajes lógico-funcionales con semántica operacional correcta y completa están basados en *narrowing*, una combinación del principio de reducción de los lenguajes funcionales junto con el principio de resolución de los lógicos.

El objetivo es extender el paradigma de la programación lógica inductiva a lenguajes lógico-funcionales dado que la mayor expresividad de estos lenguajes se puede aprovechar también para la inducción. Por ejemplo, la programación lógico-funcional inductiva (IFLP) permite un tratamiento más natural de los problemas de clasificación (es decir, estimación de una *función* discreta) que tienen que simularse en ILP o bien como predicados utilizando artificialmente modos de entrada-salida, o bien los algoritmos de inducción se deben rehacer a este efecto (véase por ejemplo el paso de [11] a [13]). Del mismo modo, algunos sistemas ILP utilizan tipos y esquemas expresados en metalenguajes diferentes de la lógica de primer orden, cosa que no sería necesaria con extensiones de orden superior de los lenguajes lógico-funcionales. Finalmente, hay otras restricciones 'de facto'; por ejemplo, pocos sistemas de ILP funcionan con predicados no aplanados, hecho que es intrínseco en el caso de IFLP, y además, en este caso es posible abordar problemas con datos semi o no estructurados.

En [6] se presentó un marco general y "fuertemente completo" para la inducción de programas lógico-funcionales. La propiedad de ser "fuertemente completo" indica la capacidad de inducir todos los posibles programas que cubran todos los ejemplos positivos sin cubrir ningún ejemplo negativo. En este marco, la evidencia se compone de ecuaciones sin variables cuyas partes derechas están en forma normal con respecto a un conocimiento previo y a la teoría a ser inducida.

En este trabajo presentamos el sistema FLIP, una implementación pragmática del marco de programación lógico-funcional inductiva, así como una serie de experimentos realizados con el mismo.

2 Programación lógico-funcional

2.1 Preliminares

En esta sección recordamos las nociones básicas relacionadas con la programación lógico-funcional.[1] Consideramos una *signatura* Σ dividida en un conjunto $\mathcal{C}$ de *constructores* y un conjunto $\mathcal{F}$ de símbolos de función *definidos*. $\mathcal{X}$ denota un conjunto infinito y contable de variables. $\mathcal{T}(\Sigma, \mathcal{X})$ representa el conjunto de términos construidos desde Σ y $\mathcal{X}$. Denotamos por $\mathcal{V}ar(t)$ el conjunto de variables que aparecen en el término t. Considerando la representación usual de un término como un árbol etiquetado, definimos el concepto de *ocurrencia* como una secuencia de enteros que denotan un camino de acceso en un término, e.g., el subtérmino de $\mathtt{f(a, g(X))}$ en la posición 1 es $\mathtt{a}$, mientras que la posición 2.1 denota el subtérmino variable $\mathtt{X}$. La posición más externa del término se indica con la ocurrencia Λ. $O(t)$ denota el *conjunto de ocurrencias* de t. $t|_p$ representa el subtérmino de t en la posición p, y $t[s]_p$ el resultado de reemplazar en t dicho subtérmino por s. Una *sustitución* σ se representa por $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, donde $\sigma(x_i) = t_i$, $i = 1, \ldots, n$, y $\sigma(x) = x$ para el resto de variables. La sustitución identidad se representa por id. Un término s es una *instancia* de un término t si existe una sustitución σ tal que $s = \sigma(t)$. La composición de dos sustituciones δ y σ se denota por $(\delta \circ \sigma)(x)$ para todo $x \in \mathcal{X}$. Escribimos $\sigma \leq \sigma'$ sii existe una sustitución δ tal que $\sigma' = \delta \circ \sigma$. Decimos que una sustitución θ es un *unificador* de dos términos s y t si $\theta(s) = \theta(t)$. σ es un unificador más general (mgu) si es un unificador y $\sigma \leq \sigma'$ para cualquier otro unificador σ'.

En la programación lógico-funcional se consideran los programas como *sistemas de reescritura de términos* (SRT en lo sucesivo), esto es, un conjunto de reglas de reescritura $l \rightarrow r$ (ecuaciones l=r orientadas de izquierda a derecha), donde l es la parte izquierda de la regla (lhs) y r la parte derecha (rhs). Dado un SRT $\mathcal{R}$, decimos que t se reescribe a s en un paso si existe una regla $l \rightarrow r \in \mathcal{R}$, una ocurrencia $p \in O(t)$ y una sustitución σ tal que $t|_p = \sigma(l)$ (t y l

[1] En [1, 4] se puede encontrar una descripción de todas las nociones técnicas involucradas.

emparejan) y $s = t[\sigma(r)]_p$. La relación $\rightarrow$ recibe el nombre de relación de reescritura. Un término está en *forma normal* (o está *normalizado*) si no es posible aplicar sobre él ningún paso de reescritura. Un SRT $\mathcal{R}$ es *terminante* si no existe ninguna derivación de reescritura infinita con respecto a $\mathcal{R}$, y es *confluente* si para todo $t, t_1, t_2 \in \mathcal{T}(\Sigma, \mathcal{X})$ con $t \rightarrow^*_{\mathcal{R}} t_1$ y $t \rightarrow^*_{\mathcal{R}} t_2$, entonces existe un elemento $t_3 \in \mathcal{T}(\Sigma, \mathcal{X})$ tal que $t_1 \rightarrow^*_{\mathcal{R}} t_3$ y $t_2 \rightarrow^*_{\mathcal{R}} t_3$. Si $\mathcal{R}$ es confluente y terminante, entonces se dice que es *canónico*.

2.2 Narrowing

Narrowing es el principio operacional más extendido para los lenguajes lógico-funcionales. Narrowing es un método constructivo que consiste en encontrar posiciones reducibles que puedan unificarse con una parte izquierda de una regla de un SRT para obtener una sustitución (mgu). La principal diferencia con la reescritura es que la sustitución puede instanciar variables tanto en la regla como en el término a reducir (variables lógicas) reemplazando el emparejamiento por la unificación.

Formalmente, dado un programa lógico funcional P se dice que un término t se reduce mediante narrowing a un término t', y se denota como $t \rightsquigarrow_\sigma t'$, si: (1) p es una posición no variable en t (i.e, $t|_p \notin \mathcal{X}$), (2) $l = r$ es una nueva variante de una regla de P, (3) la sustitución σ es un mgu de $t|_p$ y l y (4) $t' = \sigma(t[r]_p)$. Narrowing es un mecanismo de deducción completo y correcto para la clase de programas confluentes y terminantes [4].

La dificultad de una derivación de narrowing estriba en la aplicación de la regla conveniente en el término apropiado de un objetivo. El método de narrowing por sí solo explota un amplio espacio de búsqueda y muchos caminos infinitos, incluso para programas simples. Con el fin de usar narrowing como una semántica operacional práctica, se han definido un conjunto de estrategias que reducen en gran medida el espacio de búsqueda original manteniendo la propiedad de corrección y completitud bajo ciertas restricciones [4]. Sin embargo, el uso de estrategias no es directamente extrapolable si el procedimiento de narrowing se utiliza como regla inductiva. En este caso, el interés principal es generar todas las posibles ecuaciones sin descartar ninguna a priori, por lo que trabajaremos con narrowing completo.

3 Inducción de programas lógico-funcionales

En esta sección describimos el marco de programación lógico-funcional inductiva presentado en [9], así como su implementación en el sistema FLIP. El algoritmo parte de las evidencias positiva E^+ y negativa E^-. En los casos en los que se disponga se puede añadir un conocimiento previo (un programa lógico-funcional) denotado por B.

A partir de estos datos de entrada se pretende inducir un programa final P tal que $B \cup P \models E^+$ y $B \cup P \not\models E^-$. E^+ y E^- están formados por ecuaciones con su parte derecha normalizada con respecto a B y la teoría P, con $B \cup P$ canónico.

3.1 Generación de CRG

El algoritmo básico para la IFLP aprende programas generando dos conjuntos de hipótesis: un conjunto de ecuaciones (EH, ecuaciones hipótesis) donde las ecuaciones son principalmente generadas por un proceso de generalización, y un conjunto de programas (PH, programas hipótesis) compuestos exclusivamente de ecuaciones de EH. En esta sección mostramos cómo el sistema FLIP calcula las generalizaciones.

Definición 3.1 Generalización Restrictiva (RG)
Dada una ecuación $e \equiv \{t = s\}$, *la ecuación* $t' = s'$ *es una generalización restrictiva de* e *si es una generalización de* e *(i.e.*$\exists\theta : \theta(t') = t \wedge \theta(s') = s$*) tal que* $\forall x (x \in Var(s') \Rightarrow x \in Var(t'))$ *y* $t' \notin \mathcal{X}$.

Informalmente, una RG es cualquier generalización que no introduce variables extra en la parte derecha de las ecuaciones y descarta adicionalmente las ecuaciones cuya parte izquierda es una variable (es decir, las ecuaciones triviales de la forma $X = rhs$ no se consideran ya que conllevan implícitamente problemas de no terminación). RG realiza por lo tanto una primera poda en el conjunto global de generalizaciones, descartando aquellas que no deben considerarse en el proceso de inducción.

El conjunto de generalizaciones restrictivas debe refinarse aún más para considerar sólo aquellas generalizaciones que son consistentes con los ejemplos positivos y negativos. Formalmente:

Definición 3.2 Generalización Consistente Restrictiva (CRG)
Una ecuación $e' \equiv \{l_1 = r_1\}$ es una generalización consistente restrictiva (CRG) de e con respecto a E^+ y E^- y una teoría $T = B \cup P$ sii: (1) e' es una RG de e, (2) no existe ninguna derivación de narrowing usando e' y T que tenga éxito con alguna ecuación de E^- (consistencia negativa), (3) para toda ecuación $l = r \in E^+$, no se computa una forma normal de l diferente de r con respecto a $T \cup e'$ (consistencia positiva).

En FLIP, para la generación eficiente de todas las CRG's de una ecuación, hacemos uso del conjunto de ocurrencias de las ecuaciones, produciendo un árbol de ocurrencias etiquetado con clases de equivalencia. Estas clases se establecen a partir de las ocurrencias.

En la Figura 1 se muestra el árbol de ocurrencias etiquetado para la ecuación $e \equiv \{sum(0, s(0)) = s(0)\}$. Las ramas se han etiquetado con la ocurrencia de e analizada en cada paso. Los nodos contienen las clases de equivalencia las cuales se denotan por números (comenzando por 1). Así, la clase 1 denota el término $sum(0, s(0))$, la clase 2 denota $s(0)$, y así sucesivamente.

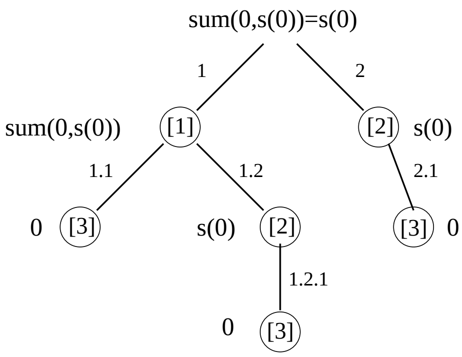

Figura 1: Árbol de ocurrencias etiquetado de la ecuación sum(0,s(0))=s(0)

Cada clase de equivalencia tiene asociado un almacén de variables. Este almacén en principio está vacío, y se va rellenando con las variables utilizadas por cada clase, según se vayan generando las ecuaciones CRG.

La generación de las CRG's se efectúa a través de un algoritmo recursivo que recibe de entrada la ecuación a generalizar, el árbol de ocurrencias y la estructura que contiene todos los almacenes. Para calcular todas y cada una de las generalizaciones, el algoritmo recorre el árbol de ocurrencias primero en profundidad y de izquierda a derecha. Las generalizaciones serán cada una de las hojas del árbol. En cada nodo se explota una ocurrencia del término que no haya sido previamente considerada. Los nodos hijos se obtienen reemplazando el término por una variable nueva (rama izquierda), y por las variables que aparecen en el almacén correspondiente (resto de ramas). Finalmente, la rama de la derecha explota el resto de ocurrencias no considerandas en el nodo padre. Por ejemplo, para la ecuación $sum(0, 0) = 0$, la primera ocurrencia sería $sum(0, 0)$, y como el almacén está vacío se reemplaza el término por una variable nueva X que se añade al almacén de la ocurrencia 1 (rama izquierda del árbol de la Figura 2). El subárbol derecho analiza las otras ocurrencias de la ecuación $sum(0, 0) = 0$.

Este proceso genera todas las generalizaciones de una ecuación por lo que para obtener las CRG debemos realizar una serie de filtros para descartar las ecuaciones no válidas según la definición 3.2.

La Figura 2 muestra el árbol de RG generado para una regla ejemplo. Las reglas señaladas con un punto son las que pasan el filtro de ecuaciones RG.

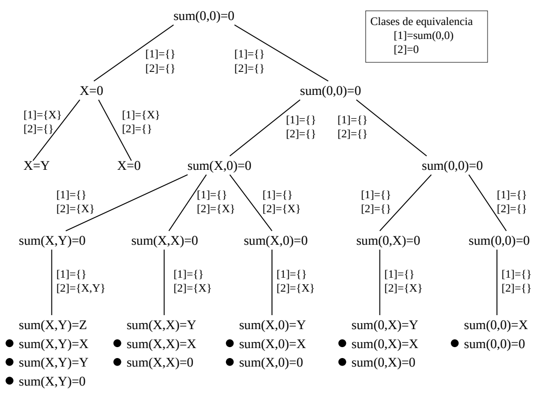

Figura 2: Árbol de RG con los almacenes de variables de la ecuación sum(0,0)=0

Una vez obtenido el conjunto de ecuaciones RG debemos comprobar la consistencia positiva y negativa. Para que una regla sea consistente con los ejemplos negativos, es necesario que falle al intentar demostrar todos los ejemplos negativos. Esta propiedad la comprobamos creando un programa con cada una de las reglas obtenidas del algoritmo anterior. A cada programa le lanzamos como objetivo cada uno de los ejemplos negativos, si un programa falla para todos los ejemplos negativos, cumple la consistencia negativa.

El caso de la consistencia positiva es un poco más complejo. Para que una regla satisfaga la consistencia positiva no debe ser contradictoria con los ejemplos positivos. Para la comprobación de esta propiedad se construye un programa con cada una de las reglas obtenidas por el algoritmo de RG y que han pasado el filtro de la consistencia negativa. A cada uno de estos programas se le lanza como objetivo los ejemplos positivos, con una pequeña modificación: la parte derecha de los ejemplos se reemplaza por una variable (ya que los ejemplos están normalizados a la derecha), por ejemplo, $sum(0,0) = 0$ se reemplaza por $sum(0,0) = Y$. Para cada ejecución de narrowing se obtendrá un conjunto de soluciones de las que sólo se tienen en cuenta las formadas por constructores. Para todas estas soluciones, el término al que queda enlazado la variable nueva debe coincidir con el término al que ésta ha reemplazado (la parte derecha del ejemplo). En el caso de que exista una solución diferente, la regla en cuestión no es consistente con algún ejemplo positivo por lo que es eliminada.

La Tabla 2 ilustra el proceso de eliminación de reglas que no cumplen con la definición de CRG para una regla ejemplo. La primera columna muestra las reglas que se obtienen por generalización. Si se aplica el filtro de RG quedan las reglas de la segunda columna. La última columna refleja las reglas que cumplen la consistencia positiva y la negativa con respecto a los conjuntos E^+ y E^- indicados en la Tabla 1.

Ejemplos Positivos E^+	Ejemplos Negativos E^-
sum(0,0)=0	sum(s(0),0)=0
sum(s(0),s(0))=s(s(0))	sum(0,0)=s(0)
sum(0,s(0))=s(0)	sum(s(0),s(0))=s(0)
sum(s(s(0)),0)=s(s(0))	sum(s(0),0)=s(s(0))

Tabla 1: Reglas Ejemplo

Reglas	Generalización Restrict. RG	Consistencia Negativa
sum(0,0) = 0	sum(0,0) = 0	sum(0,0) =0
sum(0,0) = X	sum(0,X) = 0	sum(0,X) = 0
sum(0,X) = 0	sum(0,X) = X	sum(0,X) = X
sum(0,X) = X	sum(X,0) = 0	sum(X,0) = X
sum(0,X) = Y	sum(X,0) = X	sum(X,X) = 0
sum(X,0) = 0	sum(X,Y) = 0	
sum(X,0) = X	sum(X,Y) = X	Consistencia Positiva
sum(X,0) = Y	sum(X,Y) = Y	
sum(X,Y) = 0	sum(X,X) = 0	sum(0,0)=0
sum(X,Y) = X	sum(X,X) = X	sum(0,X) = X
sum(X,Y) = Y		sum(X,0) = X
sum(X.Y) = Z		
sum(X,X) = 0		
sum(X,X) = X		
sum(X,X) = Y		
X = 0		
X = Y		

Tabla 2: Selección de reglas CRG para la ecuación sum(0,0)=0

Una vez hemos eliminado las reglas no consistentes, comparamos el conjunto de reglas CRG generadas para ese ejemplo con las reglas ya existentes (en el caso de que existan) provenientes de ejemplos anteriores. De esta manera, aunque el propio algoritmo RG no produce reglas repetidas partiendo de un ejemplo, evitaremos que existan dos reglas repetidas generadas de diferentes ejemplos.

Tras este último paso tendremos el conjunto de reglas que cumple con la definición de CRG para todos los ejemplos positivos.

Describiéndolo formalmente:

GenerateCRG
Input: $Eq, ArbOc, AlmVar, EqCRG$.

begin
if $ArbOc{=}\varnothing$ {Hoja del árbol, ecuación }
then
    begin
    if filter(Eq) **then**
        $EqCRG{=}EqCRG{+}Eq$
    Exit
GenerateCRG(Eq,
    $ArbOc- > next, AlmVar, EqCRG$)
$Vaux$=**NewVrb**
Neq=**NewEq**($Eq, ArbOc, Vaux$)
$NArbOc$=**Prune**($ArbOc$)
GenerateCRG($Neq, NArbOc- > next$,
  $AlmVar, EqCRG$)
$\forall Vbl \in AlmVar[ArbOc]$
    Neq=**NewEq**($Eq, ArbOc, Vbl$)
    GenerateCRG($Neq, NArbOc- >$
        $next, AlmVar, EqCRG$)
end

La generación de reglas por CRG produce una cantidad inmensa de nuevas reglas, por lo que para optimizar recursos en el proceso sólo tomamos las N mejores (considerando mejores aquellas que cumplan los criterios de selección de la Sección 3.3), siendo N un límite fijado por el usuario.

3.2 Operadores

Para la generación de nuevos programas a partir de otros programas hemos definido una serie de operadores de combinación. Estos operadores junto con el CRG constituyen la base del proceso de inducción de programas. En el sistema FLIP se han incluido dos: *unión* y *narrowing inverso*. Estos dos operadores tienen características opuestas, pero su combinación posibilita que se puedan generar una gran cantidad de programas simples.

3.2.1 Unión

La aplicación sobre dos programas del operador unión consiste en tomar cada una de las reglas de los dos programas juntándolas en un solo programa. Por lo tanto, el programa resultante cubrirá un número de ejemplos igual o superior a la suma de los ejemplos cubiertos por cada uno de los programas constituyentes por separado.

Este operador permite la incorporación al programa resultante de las reglas simples que no tienen llamadas a funciones en su parte derecha. Estas reglas constituyen en muchas ocasiones el caso base de las funciones cuya definición se está induciendo. Por ejemplo, $sum(X, 0) = X$ en el programa que define la suma, o $member(p(X, Y), Y) = true$ en el programa que define la función booleana "pertenece a una lista".

Sin embargo el uso en exceso de este operador provoca que se generen programas con gran cantidad de reglas que sólo sirvan para cubrir los ejemplos concretos de una muestra en cuestión, sin llegar al programa general.

3.2.2 Narrowing inverso

El operador de narrowing inverso se deriva directamente de la inversión del mecanismo de narrowing rebajando la condición de aplicar una regla sólo a una ocurrencia no variable. Su funcionamiento se ilustra con el siguiente ejemplo:

Ejemplo 3.3 *Supongamos que estamos induciendo un programa P de los ejemplos positivos de la Tabla 1 y que hemos generado todos los CRG. En el paso n, supongamos que se selecciona la regla $X + 0 = X$ como válida para P y seleccionamos arbitrariamente la parte derecha de la regla $X + s(0) = s(X)$, i.e., $s(X)$. La primera regla puede ser usada inversamente en el segundo término en varias ocurrencias. En este caso concreto, existen dos posibles aplicaciones correspondientes a una ocurrencia variable (que denota el subtérmino X) y a otra no variable (que denota el subtérmino $s(X)$), resultando:* ($\mathtt{t}_1$) $s(X + 0)$ *y* ($\mathtt{t}_2$) $s(X) + 0$. *De ahí se concluye que $\mathtt{t}_1$ y $\mathtt{t}_2$ pueden derivarse mediante narrowing a $s(X)$ usando la primera regla. Las ecuaciones resultado de este proceso son $X + s(0) = s(X + 0)$ y $X + s(0) = s(X) + 0$.*

De manera informal se puede explicar el proceso de la siguiente forma: dadas dos reglas, se le da la vuelta a la primera (o regla emisora) y se realiza un paso de narrowing a cada una de las ocurrencias de la parte derecha de la segunda regla (o regla receptora). De esta forma se obtienen tantos términos como ocurrencias a las que se pueda aplicar la regla emisora. Para cada uno de estos términos se construye una nueva regla, formada por estos términos en la parte derecha y la parte izquierda de la regla receptora en su parte izquierda. Formalmente:

Definición 3.4 Narrowing inverso
Dado un programa lógico-funcional P, decimos que un término t deriva mediante narrowing inverso en un término t', y escribimos $t \overset{u,l=r,\theta}{\hookleftarrow}_P t'$, si y sólo si $u \in O(t)$, $l = r$ es una nueva variante de la regla de P, $\theta = mgu(t_{|u}, r)$ y $t' = \theta(t[l]_u)$. A la relación $\hookleftarrow_P$ se le llama narrowing inverso.

Este operador permite construir reglas complejas que incluyan llamadas a funciones en su parte derecha. Por lo tanto posibilita el aprendizaje de reglas recursivas a partir de los ejemplos generalizados tal y como se ilustra en el Ejemplo 3.3.

Uno de los aspectos más importantes para el funcionamiento eficiente del operador de narrowing inverso se encuentra en la elección adecuada de las reglas a las cuales aplicar el operador. En el sistema FLIP lo aplicamos de la siguiente forma: dados dos programas tomamos las dos mejores reglas de cada programa. La regla elegida del mejor programa actúa de regla emisora, por lo que se aplica a las ocurrencias de la parte derecha de la regla receptora. En el caso de que esta combinación no genere ninguna regla consistente, se intenta con la combinación inversa.

Un problema que se puede derivar de la aplicación de este operador es la aparición de reglas con variables extra en su parte derecha (ver Ejemplo 3.5). Si se procediera al descarte de este tipo de reglas durante el proceso se perderían ecuaciones que podrían conducir a la solución final, por lo que es necesario un postproceso de dichas reglas para eliminar las variables extra. Esto se consigue instanciando las variables extra con cada una de las variables de la parte izquierda de la ecuación. Esta instanciación se efectúa de forma que se den todas las posibilidades sin que dos variables extras diferentes se asignen con la misma variable de la parte izquierda. Por lo tanto, si tenemos n variables en la parte izquierda y m variables extra en la parte derecha se obtendrán P_n^m, es decir $\frac{n!}{(n-m)!}$, ecuaciones diferentes. En el caso de que haya mayor número de variables extra que variables a asignar no se generan ecuaciones por narrowing inverso.

Ejemplo 3.5 *Supongamos que deseamos inducir el programa member que determina si un elemento pertenece a una lista. Si se selecciona para la aplicación del narrowing inverso las siguientes reglas consistentes,* $member(p(X,Y),Y) = true$ *y* $member(p(p(X,Y),Z),Y) = true$ *(reglas emisora y receptora respectivamente), se obtendría* $member(p(p(X,Y),Z),Y) = member(p(A,B),B)$. *Esta regla contiene variables extra en su parte derecha por lo que éstas se reemplazan por las variables de la parte izquierda, resultando las siguientes ecuaciones:*

$$member(p(p(X,Y),Z),Y) = member(p(X,Y),Y)$$
$$member(p(p(X,Y),Z),Y) = member(p(X,Z),Z)$$
$$member(p(p(X,Y),Z),Y) = member(p(Y,X),X)$$
$$member(p(p(X,Y),Z),Y) = member(p(Y,Z),Z)$$
$$member(p(p(X,Y),Z),Y) = member(p(Z,X),X)$$
$$member(p(p(X,Y),Z),Y) = member(p(Z,Y),Y)$$

De la primera ecuación y mediante CRG, se obtendría la ecuación $member(p(X,Y),Z) = member(X,Z)$, *que junto con la regla* $member(p(X,Y),Y) = true$ *forman el programa solución.*

Otra circunstancia que puede darse es que se generen ecuaciones con carácter no terminante, como $mod2(s(X)) = mod2(s(s(s(X))))$. Por ello cada vez que se genera una ecuación mediante narrowing inverso se comprueba si dicha regla posee el mismo símbolo de función más externo en ambas partes de la ecuación y si el tamaño de la parte izquierda es menor que la parte derecha; si es así, se invierte la ecuación.

Una vez se han generado las nuevas reglas con narrowing inverso se construye por cada una de ellas un conjunto de reglas CRG.

Por cada una de las reglas resultantes se genera un programa que consta de la regla resultante, la regla emisora, y las reglas restantes de los dos programas iniciales que no han tomado parte en el proceso de narrowing inverso. En esta composición de ecuaciones se comprueba que no se repitan reglas idénticas.

3.3 Bucle principal

Una vez ya hemos generado el conjunto de CRG con cada uno de los ejemplos positivos y vistos los operadores, podemos describir el proceso global.

Primeramente construimos un programa por todas y cada una de las reglas CRG obtenidas a partir de la evidencia positiva; llamaremos a este grupo de programas, conjunto de programas iniciales. Calculamos para estos programas la cobertura y la optimalidad. Estos parámetros sirven para establecer unos criterios que permitan comparar entre sí diferentes programas. Concretamente la cobertura (Cov) se define como el conjunto de ejemplos positivos que un programa demuestra y $CovF^+(P)$ se define como $card(Cov(P))$. La optimalidad puede calcularse a partir de la cobertura y de la longitud del programa, en clara referencia al principio MDL [8]. La longitud de un programa se define como $LenF(P) = -\sum_{e \in P} log_2 tam(e)$, donde $tam(e) = 1 + n_v/2 + n_c + n_f$ siendo n_v, n_c y n_f el número de variables, de constantes y de funciones de la parte derecha de las reglas respectivamente. De esta forma definimos la optimalidad[2] como $Opt(P) = \alpha \times LenF(P) + \beta \times CovF^{+}(P)$.

Para conseguir programas con mayor optimalidad es necesario combinar diferentes programas iniciales mediante los operadores de la Sección 3.2. Como el orden de combinaciones es cuadrático, es necesario empezar a ensayar aquellos programas que son mejores. Con el concepto de mejor, referimos al programa que cubre más ejemplos positivos con mayor optimalidad. Para fomentar que los dos mejores programas sean complementarios (es decir, que cubran ejemplos distintos a ser posible) podríamos combinar entre sí cada par de programas uniéndolos, generando un nuevo programa compuesto. Sin embargo el hecho de calcular la cobertura de todos y cada uno de estos programas compuestos para ser evaluados supone un coste muy elevado, ya que se debe lanzar un proceso de narrowing por cada programa y ejemplo. Si existieran n programas iniciales se deberían realizar C_n^2, o sea $\frac{n^2-n}{2}$ ejecuciones de narrowing por cada ejemplo positivo. Si hay m ejemplos positivos, tendríamos entonces $\mathcal{O}(\frac{n^2*m}{2})$ operaciones de narrowing. Para evitar este procedimiento tan costoso introducimos una pequeña simplificación que consiste en no calcular la cobertura exacta de un programa compuesto sino en la unión de las coberturas de los programas que lo forman. Esta aproximación simple estará siempre por debajo del valor real, ya que un programa formado por dos reglas aceptará los ejemplos que acepten cada una de las reglas por separado, pero en algunos casos podrá aceptar más ejemplos. Cabe reseñar que la unión de las coberturas de los ejemplos se puede interpretar como una suma lógica (OR). Por ejemplo, si el programa $P1$ cubre los ejemplos $e1$, $e2$ y $e4$, es decir $Cov(P1) = \{e1, e2, e4\}$ y el programa $P2$ tiene como cobertura $Cov(P2) = \{e1, e3, e4\}$, con nuestra aproximación $Cov(P1 \cup P2) \approx Cov(P1) \cup Cov(P2) = \{e1, e2, e3, e4\}$.

Para ello utilizamos en FLIP una estructura asociada a cada regla que tiene como longitud en bits el número de ejemplos positivos, donde se marca con 1 ó 0 si el ejemplo se cubre o no, respectivamente. Esta estructura, además de minimizar el tamaño necesario para almacenar la información sobre la cobertura, nos será perfecta para efectuar la suma lógica en nuestra aproximación del cálculo de la cobertura de los programas compuestos. Antes de iniciar el bucle necesitamos calcular los ejemplos que cubre cada una de las uniones de los programas simples. Con esta implementación realizamos del orden de $\mathcal{O}(\frac{n^2*m}{2})$ operaciones OR de un solo bit en cada iteración para obtener el mejor de los programas (en lugar de operaciones de narrowing).

Cada vez que comparamos uno a uno los programas iniciales para seleccionar el mejor programa compuesto almacenamos el número de ejemplos que éste cubre, tal como hemos descrito. Necesitaremos por tanto una matriz triangular de n elementos, siendo n el número de programas iniciales. La función de esta estructura, que denotamos como mapa de programas, es tener almacenada la información sobre las combinaciones entre los programas de manera que se evite la repetición de la misma combinación de dos programas.

Seleccionamos el mejor programa compuesto como aquel que tenga mayor cobertura aproximada. En el caso de que dos o más programas compuestos tengan la misma cobertura, se elige el que tenga mejor optimalidad.

Para calcular la optimalidad actuamos de manera similar. Teóricamente, para el cálculo de la optimalidad de un programa es necesario saber la cobertura de dicho programa, por lo que si deseamos saber la optimalidad exacta de cada uno de los programas compuestos que se comparan resultaría un proceso con un coste altísimo. Es por ello que también utilizamos en FLIP una aproximación para el cálculo de la optimalidad. Esta aproximación consiste en tomar la optimalidad del programa compuesto como la suma de las optimalidades de los dos programas simples que lo forman. De esta forma se ahorra gran parte del tiempo de ejecución del proceso de selección del mejor programa.

Todo este proceso de selección del mejor programa compuesto tiene como objetivo distinguir los dos mejores programas simples que lo forman.

Una vez seleccionado el mejor par de programas simples, aplicamos cada uno de los operadores de la Sección 3.2 a los mismos, dando lugar a uno o varios programas nuevos por cada opera-

[2] En el sistema FLIP consideramos $\alpha = 1$ y $\beta = 1$.

dor aplicado. Con el fin de no volver a generar el mismo programa en pasos posteriores, marcamos la intersección de sus programas constituyentes en el mapa de programas con un 0.

Tras esto entramos en el bucle principal. Fundamentalmente en el bucle se repite el proceso que hemos explicado, pero en este caso sólo se procede a la comparación de los nuevos programas con los ya existentes. Con esto se añade una fila (y una columna) al mapa de programas por cada nuevo programa. Una vez rellenado el mapa, seleccionamos el mejor programa con el método anteriormente explicado, se calcula su cobertura y optimalidad exactas, y se marca la intersección de los programas que lo forman en el mapa de programas. A continuación se inicia una nueva iteración del bucle. De esta manera el algoritmo tiene un coste del orden de $\mathcal{O}(\frac{n^2*m}{2})$ para la primera iteración, y $\mathcal{O}(n*m)$ a partir de la segunda, siendo n el número de programas y m el número de ejemplos positivos.

El criterio de parada del bucle lo establecemos cuando se selecciona un programa que cubra todos los ejemplos con una optimalidad por encima de una constante establecida, o se llegue a un número de iteraciones límite.

3.4 Aprendizaje con conocimiento previo

Una de las grandes ventajas de la ILP sobre otros paradigmas de aprendizaje es que su gran expresividad e inteligibilidad facilitan el uso de conocimiento previo en la resolución de problemas de inducción que serían prácticamente intratables sin la especificación de estas funciones auxiliares (véase, e.g. [14]). Piénsese, por ejemplo, en inducir la función potencia sin conocer el producto, o inducir la función de parentesco sin la definición de padre y madre.

Aparte de estas ventajas, el uso de conocimiento previo introduce también otros inconvenientes: el número de posibles combinaciones de dónde y cómo utilizar el conocimiento previo se dispara y se debe recurrir a heurísticas [11].

A continuación presentamos el modo en que FLIP introduce funciones del conocimiento previo en los programas que están siendo inducidos.

3.4.1 Narrowing inverso con conocimiento previo

Este método consiste en la realización de pasos de narrowing inverso con respecto a funciones del conocimiento previo. Normalmente seleccionamos (mediante la preferencia de partes derecha más cortas) los casos base de las funciones del conocimiento previo ya que éstas tienen la parte derecha simple por lo es más sencillo su utilización para el narrowing inverso. De esta manera conseguimos que las funciones del conocimiento base sean tomadas en cuenta durante la inducción.

Por ejemplo, si la función objetivo es el producto de dos números naturales y usamos un conocimiento previo que contenga la definición de la suma entre dos naturales, entonces la ecuación $prod(X,0) = 0$ es completamente equivalente a $prod(X,0) = sum(0,0)$ (ya que $sum(X,0) = X$ es una regla de B). Sin embargo, esta última ecuación contiene el símbolo sum del conocimiento previo.

En adelante denotaremos por BF (funciones básicas) el conjunto de funciones de B, determinadas por el usuario, que pueden ser utilizadas en la definición de las funciones aprendidas.

3.5 Algoritmo de IFLP

El proceso principal de inducción de programas lógico-funcionales se formaliza con el siguiente algoritmo que trabaja con un conjunto de reglas (denotado por EH) y con un conjunto de programas (denotado por PH) formado exclusivamente de reglas pertenecientes a EH. En cada paso del algoritmo se generan nuevas ecuaciones y programas por narrowing inverso y por unión:

Input: E^+, E^-, B, BF.
Output: Un programa P
begin
Let $EH = \varnothing$ and let $PH = \varnothing$
GenerateCRG(input:$E^+, E^-, \varnothing$; output: EH)
$PH = \{\{e\}/e \in EH\}$
Let $BestSolution = Select\_best(PH)$
while not $stop\_criterion(BestSolution)$ **do**
    if using B
    then begin
    for each $e \in EH$ **do**
        Let $P = \{e\}$
        $InverseNarrowing$(input:P, B, BF;
                                output:EH', PH')

$Update\_all(BestSolution,$
$EH, PH, EH', PH')$

endfor
endbegin
endif { Uso del conocimiento previo}
{Caso general. Selección del mejor par de programas P_1, P_2 que no han sido seleccionados previamente de PH}
Let $n = card(E^+)$
while $n > 0$ **do**

$PP = \{(P_1, P_2) \mid P_1, P_2 \in PH$ and not $marked[P_1, P_2], P_1 \neq P_2$ s.t. $card(\{e \in E^+ \mid P_1 \models e \vee P_2 \models e\}) \geq n\}$
if $PP \neq \varnothing$
then

let $(P_1, P_2) = argmin_{PP}(Opt(P_1) + Opt(P_2))$;
break while

else let $n = n - 1$
endif

endwhile
$marked[P_1, P_2]$= true;
if $n = 0$ **then begin**

$GenerateCRG$(input:E^+, E^-, EH;
output: EH')
if $EH' = EH$ **then halt** {No hay más programas que tratar.}
endbegin
else begin
$InverseNarrowing$(input:$P_1, P_2, \varnothing$;
output:EH', PH')
$Update\_all(BestSolution,$
$EH, PH, EH', PH')$
endbegin
endif
endwhile
endalgorithm

donde:

- $Select\_best(PH)$ selecciona el programa con la mejor cobertura y, en caso de igualdad, la mejor optimalidad.
- $Update\_all(S, E, P, E', P')$ realiza las siguientes acciones: $E = E \cup E'$, $P = P \cup P'$ y $S = Select\_best(P)$.
- $marked[P, P']$ es $true$ si P y P' han sido seleccionados para ser combinados por narrowing inverso en algún paso previo.

3.6 Aplicaciones

Hemos implementado el algoritmo de inducción de programas lógico-funcionales creando el sistema FLIP. FLIP es un programa construido en C, con más de 5.000 líneas de código. El sistema incluye un parser simple, un mecanismo para resolver ecuaciones mediante narrowing [7], un método de narrowing inverso y un generador de ecuaciones CRG.

El sistema FLIP trabaja con dos conjuntos diferentes de hechos, ejemplos positivos y ejemplos negativos y si se desea, conocimiento previo. Partiendo de estos ejemplos el sistema aplica el algoritmo IFLP hasta encontrar un programa solución o el bucle principal de inducción sobrepasa un número máximo de pasos.

Para comprobar la utilidad de nuestro marco de inducción hemos ensayado con FLIP la inducción de diferentes programas a partir de ejemplos positivos y negativos. Lógicamente, el éxito del proceso de inducción depende de la dificultad del programa a aprender y de la calidad de los ejemplos iniciales. El tamaño en número de reglas de los programas inducidos no tiene límite; sin embargo, los programas con muchas funciones necesitan más pasos del bucle de inducción por lo que son más difíciles de resolver.

Los resultados presentados en esta sección fueron generados con ejemplos de los bancos de prueba mostrados en [2]. Las evidencias son relativamente pequeñas en todos los casos, variando de 3 a 12 ejemplos positivos y de 2 a 11 ejemplos negativos.

Hemos probado el sistema con diferentes tipos de programas, el código de los programas inducidos se encuentra en [2]. La Tabla 3 muestra algunos programas no recursivos. Como puede esperarse, el mecanismo CRG es suficiente para inducir este tipo de programas, por lo que se inducen en tan sólo un paso del algoritmo. Este tipo de problemas se corresponde con el problema clásico de clasificación no recursivo. Como puede apreciarse la notación funcional es mucho más natural para expresarlos que la representación lógica, especialmente para aquellos casos donde hay más de dos clases. Los ejemplos *cup*, *enjoysport* y *playtennis* han sido extraídos de [8]. Debido a la restricción de confluencia, las reglas no pueden solaparse lo cual significa que para los problemas no recursivos, la solución siempre se puede representar como un árbol de decisión. Por ejemplo, en el problema *playtennis*, el sistema FLIP genera un árbol de decisión usando únicamente el operador CRG (Figura 3).

En los programas recursivos es donde FLIP se muestra más potente, como se evidencia en la Tabla 4. Funciones como *app* se definen de forma más natural a través de funciones que con

Programa Inducido	Atributos
cup(y,up,X1,X2,n,y,X3)=true, cup(y,up,X1,X2,side,y,X3)=true	BottomIsFlat, Concav.(up,not-up, no), Expensive, HandleOnTop, Fragile, Handle (n, top, side), Light, MadeOf(ceramic, paper,styrofoam)
sport(sunny,X,Y,Z,W,V)=yes sport(cloudy,X,Y,Z,W,V)=yes	Sky, AirTemp, Humidity, Wind, Water, Forecast
tennis(overcast,X1,X2,X3)=yes tennis(sunny,X1,normal,X3)=yes tennis(rain,X1,X2,weak)=yes	Outlook, Temperature, Humidity, Wind

Tabla 3: Programas no recursivos inducidos

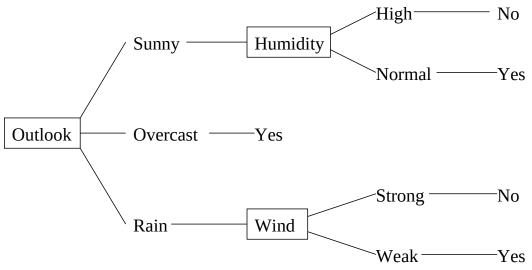

Figura 3: El árbol de decisión para el concepto Playtennis

predicados, ya que no necesita información adicional (como el uso de modos de E/S para los predicados a aprender). También destacamos la capacidad de FLIP para aprender más de una función al mismo tiempo, como en el ejemplo con *sum* y *prod*, que FLIP puede aprender conjuntamente, esto es, a partir de ejemplos de las dos funciones, FLIP induce primero *sum* y tras ello utiliza esta función para aprender *prod*.

Finalmente, mostramos en la Tabla 5 algunos programas complejos que han sido aprendidos con el uso de conocimiento previo.

A la vista de estos ejemplos, las aplicaciones potenciales del paradigma IFLP son al menos las mismas que las del paradigma ILP [10], ya que tanto los datos como el conocimiento previo expresados en lógica de primer orden se traducen a programas lógico-funcionales directamente. Por el momento, los problemas de nuestra aproximación son los mismos que los de los sistemas ILP (y de la mayoría de sistemas de aprendizaje automático): escalabilidad temporal y espacial, lo que permitiría abordar otro tipo de aplicaciones (minería de datos). No obstante, IFLP permite una aproximación más directa a la extracción de conocimiento a partir de información semiestructurada, como se detalla en [3].

Programa inducido
sum(s(X),Y) = s(sum(X,Y)) sum(0,Y)=Y
length(p(X,Y)) = s(length(X)) length(λ)=0
consec(p(X,Y)) = consec(X) consec(p(p(X,Y),Y)) = true
drop(0,X) = X drop(s(X),p(Y,Z)) = drop(X,Y)
app(p(X,Y),Z) = p(app(X,Z),Y) app(λ,X) = X
member(p(X,Y),Z) = member(X,Z) member(p(X,Y),Y) = true
last(p(X,Y)) = last(X) last(p(λ,X)) = X
geq(s(X),s(Y)) = geq(X,Y) geq(X,0) = true
sum(s(X),Y) = s(sum(X,Y)) sum(0,Y)=Y prod(s(X0),X1) = sum(prod(X0,X1),X1) prod(0,X0) = 0
mod3(0) = 0 mod3(s(0)) = s(mod3(0)) mod3(s(s(0))) = s(s(mod3(0))) mod3(s(s(s(X0)))) = mod3(X0)
even(s(s(X)) = even(X) even(0) = true

Tabla 4: Programas recursivos inducidos sin conocimiento previo.

4 Conclusiones y Trabajo Futuro

Se ha presentado un sistema que integra diferentes técnicas de la programación lógico-funcional, la programación lógica inductiva y el aprendizaje automático: narrowing, inducción por generalización, inversión de operadores deductivos, técnicas evolutivas y el principio MDL. El resultado es un sistema de aprendizaje multipropósito que genera teorías fácilmente comprensibles a partir de unos ejemplos y un conocimiento previo, y que son también muy sencillas de especificar, al estar todo (entradas y salidas) basado en reglas (ecuaciones). De este modo, el sistema FLIP permite la inducción de reglas proposicionales sobre atributos, reglas de clasificación, árboles de decisión, problemas con más de una función, teorías recursivas y puede sacar partido de la información del conocimiento previo.

Programas inducidos	B
rev(p(X0,X1)) = app(rev(X0),p(v,X1)) rev(v) = v	append
suml(p(X0,X1)) = sum(suml(X0),X1) suml(p(v,X0)) = X0	sum
maxl(p(X0,X1)) = max(X1,maxl(X0)) maxl(p(v,X0)) = X0	max
prod(s(X0),X1) = sum(prod(X0,X1),X1) prod(0,X0) = 0	sum
fact(s(X0)) = prod(fact(X0),s(X0)) fact(0) = s(0)	prod
sort(p(X0,X1)) = inssort(X1,sort(X0)) sort(v) = v	inssort

Tabla 5: Programas inducidos usando conocimiento previo

Con respecto a los objetivos de extensión del paradigma ILP a lenguajes lógico-funcionales, el sistema propuesto supone un paso a medio camino en este objetivo. Aunque las capacidades de FLIP son similares a otros sistemas ILP, y consigue tratar de una manera más natural problemas con estructuras complejas (listas, árboles, términos XML, etc.) que otros sistemas planos como FOIL ([11][13]), todavía no aprovecha todo el potencial de los lenguajes lógico funcionales: distintas estrategias deductivas, orden superior, reglas condicionales y aproximaciones a terminación.

Actualmente estamos trabajando en una versión incremental del algoritmo que, a la vista de los primeros resultados, mejora la eficiencia del mismo aproximadamente diez veces sobre el original, y una construcción más eficiente de los CRG's, utilizando la información de tipos y permitiendo valores continuos, basada en el algoritmo C4.5 de Quinlan [12]. Para una información actualizada de este desarrollo, se puede consultar la página web del sistema [2].

Referencias

1. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP system homepage. `http://www.dsic.upv.es/~jorallo/flip/`, 2000.
3. C. Ferri, J. Hernández, and M.J. Ramírez. Learning functional logic classification concepts from databases. In *9th International Workshop on Functional and Logic Programming WFLP'00*. S.U.P.V, 2000.
4. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583–628, 1994.
5. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `www.informatik.uni-kiel.de/~curry`, 2000.
6. J. Hernández and M.J. Ramírez. A Strong Complete Schema for Inductive Functional Logic Programming. In *Proc. of the Ninth International Workshop on Inductive Logic Programming, ILP'99*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 116–127, 1999.
7. P. Julián and G. Moreno. Nuevas Semánticas para Programas Lógicos Funcionales. Technical report, Departamento de Sistemas Informáticos y Computación , Universidad Politécnica de València, 1987.
8. Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
9. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.
10. S. Muggleton. Inductive logic programming: Issues, results, and the challenge of learning language in logic. *Artificial Intelligence*, 114(1–2):283–296, 1999.
11. J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.
12. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
13. J. R. Quinlan. Learning first-order definitions of functions. *JAIR*, 5:139–161, 1996.
14. M. Turcotte, S.H. Muggleton, and M.J.E. Sternberg. The effect of relational background knowledge on learning of protein three-dimensional fold signatures. *Machine Learning*, 2000.