DEMOSTRACIÓN AUTOMÁTICA DE TEOREMAS

JOSÉ DE JESÚS LAVALLE MARTÍNEZ
FACULTAD DE CIENCIAS DE LA COMPUTACIÓN, BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA, JOSÉ ARRAZOLA RAMÍREZ,
AND JUAN PABLO MUÑOZ TORIZ
FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS, BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

Resumen. En este trabajo se desarrolla un demostrador automático de teoremas en ML basado en el sistema Gentzen. Para esto se da una breve introducción a ML. Esto es: qué es ML, cuál es la sintaxis usada en este lenguaje de programación, así como algunos ejemplos de como programar en ML. Luego se explica en que consiste formalmente Gentzen y como codificar en ML las definiciones dadas. Finalmente se dan algunos ejemplos de ejecución del programa.

1. Introducción

ML[1] es un lenguaje de programación de propósito general de la familia de los lenguajes de programación funcional, desarrollado por Robin Milner y sus colaboradores a finales de la década de 1970 en la Universidad de Edimburgo. ML es un acrónimo de *Meta Lenguaje* dado que fue concebido como el lenguaje para desarrollar tácticas de demostración en el sistema LCF.

Entre las características de ML se incluyen: alto orden, evaluación por valor, álgebra de funciones, manejo automático de memoria por medio de recolección de basura, polimorfismo parametrizado, análisis estático de tipos, inferencia de tipos, tipos de datos algebráicos, llamada por patrones y manejo de excepciones. Esta combinación particular de conceptos hace que sea posible producir uno de los mejores lenguajes actualmente disponibles. Es un lenguaje funcional fuertemente tipificado, esto es que toda expresión en ML tiene un único tipo, es un lenguaje interpretado no compilado.

Una función en ML se define poniendo la palabra reservada **fun**, seguida por los parámetros de la función, el símbolo = y finalmente el cuerpo de la función.

1.1. Ejemplo. Definir en ML la función $sucesor(x) = x + 1$ se puede hacer de las siguientes maneras:

```
fun sucesor  x = x+1;
fun sucesor1 x:int = x+1;
fun sucesor2 x = x+1:int;
fun sucesor3 x:int = x+1:int;
```

1.2. Ejemplo. La función factorial definida mediante

$$factorial(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * factorial(n-1) & \text{si } n > 0 \end{cases}$$

se puede codificar en ML por:

```
fun factorial 0 = 1
|   factorial n = n * factorial(n-1);
```

Por otro lado, como ya se dijo anteriormente ML es un lenguaje de programación fuertemente tipificado, así que debemos caracterizar mediante tipos el sistema que queremos construir. Para lo cual utilizamos el constructor de tipo **datatype**.

1.3. Ejemplo. Supongamos que deseamos construir una lista de cadenas, la cual se define recursivamente como:

- Una lista de cadenas está vacía o
- Tiene una cadena como primer elemento y un resto que es una lista de cadenas.

Expresado en ML:

```
datatype strlist = nul | ht of string * strlist;
```

Con lo anterior no sólo hemos definido el tipo de datos strlist, también hemos creado dos constructores para dicho tipo, a saber nul de aridad 0 y ht de aridad 2. Así para construir una lista de cadenas tenemos que basarnos en tales constructores, como en

```
ht("cadena1",ht("cadena2",ht("cadena3",nul)));
```

Al introducir la línea anterior, ML nos dirá que el tipo al que pertenece es a strlist.

ML también cuenta con una forma de crear alias para tipos usando **type**, pero con ésta no se define constructor alguno. Como ejemplo podemos escribir

```
type float = real;
```

En este caso float se puede usar posteriormente como un sinónimo del tipo real. Una primera forma de ocultar código en ML es usando **let** e **in** como a continuación se muestra:

```
fun invierteLista(Lista) =
  let
    fun invierteLPA(nul,pa) = pa
    |   invierteLPA(ht(h,t),pa) = invierteLPA(t,ht(h,pa))
  in
    invierteLPA(Lista,nul)
  end;
```

En este caso la función invierteLista delega su trabajo a una segunda función llamada invierteLPA. La función invierteLPA se encuentra oculta entre las palabras reservadas **let** e **in**. La palabra reservada **in** indica el fin del **let** y después de ella comienza el cuerpo de la función principal.

Los lenguajes de la familia ML se usan para el diseño y manipulación de cualquier lenguaje, de ahí su nombre *Meta Language* (compiladores, analizadores, demostradores de teoremas, etc.).

2. Implementación

Primero definiremos formalmente el lenguaje de la lógica proposicional [2][3]. Para ello debemos definir su alfabeto y dar sus reglas sintácticas de formación.

2.1. Definición. El alfabeto del lenguaje de la lógica proposicional está conformado por:

1. Paréntesis: (,);
2. Conectivos lógicos: $\neg$, $\wedge$, $\vee$, $\rightarrow$;
3. Letras proposicionales: Cualquier letra con o sin subíndices.

2.2. Definición. Los elementos del lenguaje de la lógica proposicional, que llamaremos proposiciones, se definitionnen mediante:

1. Toda letra proposicional es una proposición,
2. Si $\mathcal{A}$ y $\mathcal{B}$ son proposiciones entonces $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$ y $(\mathcal{A} \rightarrow \mathcal{B})$ también son proposiciones.

Lo cual podemos escribir, prácticamente tal cual, en ML de la siguiente manera:

```
datatype Prop = lp of string |
                nega of Prop |
                conj of Prop * Prop |
                disy of Prop * Prop |
                impl of Prop * Prop;
```

Así tenemos cinco constructores para el tipo Prop:

1. lp de aridad uno y cadenas como argumento,
2. nega de aridad uno y proposiciones como argumento,
3. conj de aridad dos y proposiciones como argumentos,
4. disy de aridad dos y proposiciones como argumentos,
5. impl de aridad dos y proposiciones como argumentos.

Definimos Sec para crear listas de proposiciones, ya que en ML está interconstruido el tipo lista y es polimorfo utilizamos **type** como alias para una lista de proposicones. Para fines prácticos pensaremos en Sec como un conjunto de proposiciones.

```
type Sec = Prop list;
```

El sistema de Gentzen [2][3] nos permite saber si una proposición es una tautología, en caso de que no sea una tautología también nos permite saber bajo que asignaciones, de valores de verdad a las letras proposicionales, la proposición es falsificable, se basa en el concepto de secuente. Un secuente es una pareja (Γ, Δ) de secuencias (o conjuntos, posiblemente vacíos) de proposiciones: $\Gamma =< A_1, ..., A_n >$, $\Delta =< B_1, ..., B_m >$ y decimos que Γ y Δ son el antecedente y el consecuente, respectivamente. Por brevedad escribiremos los secuentes como $\Gamma \rightsquigarrow \Delta$ y en las secuencias no usaremos paréntesis angulares.

Por lo tanto el constructor de tipo Secuente se define de la siguiente manera:

```
datatype Secuente = lr of Sec * Sec;
```

Las reglas de inferencia están formadas como un conjunto de secuentes premisas sobre una raya horizontal y un único secuente conclusión bajo la raya.

$$\frac{premisa_1, premisa_2, \ldots, premisa_n}{conclusion}\ nombreDeLaRegla$$

Las premisas resultan de la aplicación de la definición de la regla a la conclusión. Las reglas de inferencia se clasifican en dos categorías: las que operan sobre el antecedente ($* \rightsquigarrow$) y las que operan sobre el consecuente ($\rightsquigarrow *$): cada regla descompone a la proposición principal en subproposiciones que son colocadas ya sea en el antecedente o consecuente de las premisas, pudiendo incluso dividir un secuente (conclusión) en dos (premisas). Las premisas obtenidas después de esta operación pueden contener o no operadores lógicos, en caso de contenerlos las reglas de inferencia vuelven a ser aplicadas. Ello naturalmente permite representar a las pruebas como árboles.

2.3. Definición. El sistema de Gentzen se define a continuación:

1. Γ, Δ representan secuencias arbitrarias de fórmulas (conjuntos),
2. A, B representan proposiciones arbitrarias,
3. El único axioma es:
$$\frac{}{\Gamma, A \rightsquigarrow A, \Delta}\ Ax$$
4. Las reglas de inferencia de Gentzen son:

$$\frac{\Gamma \rightsquigarrow A, \Delta}{\Gamma, (\neg A) \rightsquigarrow \Delta}\ \neg \rightsquigarrow \qquad \frac{\Gamma, A \rightsquigarrow \Delta}{\Gamma \rightsquigarrow (\neg A), \Delta}\ \rightsquigarrow \neg$$

$$\frac{\Gamma, A, B \rightsquigarrow \Delta}{\Gamma, (A \wedge B) \rightsquigarrow \Delta}\ \wedge \rightsquigarrow \qquad \frac{\Gamma \rightsquigarrow A, \Delta \quad \Gamma \rightsquigarrow B, \Delta}{\Gamma \rightsquigarrow (A \wedge B), \Delta}\ \rightsquigarrow \wedge$$

$$\frac{\Gamma, A \rightsquigarrow \Delta \quad \Gamma, B \rightsquigarrow \Delta}{\Gamma, (A \vee B) \rightsquigarrow \Delta}\ \vee \rightsquigarrow \qquad \frac{\Gamma \rightsquigarrow A, B, \Delta}{\Gamma \rightsquigarrow (A \vee B), \Delta}\ \rightsquigarrow \vee$$

$$\frac{\Gamma \rightsquigarrow A, \Delta \quad \Gamma, B \rightsquigarrow \Delta}{\Gamma, (A \to B) \rightsquigarrow \Delta}\ \to \rightsquigarrow \qquad \frac{\Gamma, A \rightsquigarrow B, \Delta}{\Gamma \rightsquigarrow (A \to B), \Delta}\ \rightsquigarrow \to$$

Note que en el sistema de Gentzen tenemos un axioma (Ax) y dos formas de reglas de inferencia, aquellas que dado un secuente conclusión obtenemos sólo un secuente premisa llamémosles InRuUno (coloquialmente las que no bifurcan) y aquellas que dado un secuente conclusión obtenemos dos secuentes premisas llamémosles InRuDos (coloquialmente las que bifurcan), lo cual podemos definir en ML mediante:

```
datatype SistemaG = Ax of Secuente |
                    InRuUno of Secuente |
                    InRuDos of Secuente * Secuente;
```

Hasta aquí se ha caracterizado el sistema de Gentzen, es decir, hemos creado un tipo SistemaG cuyos elementos son la base para poder hacer demostraciones en lógica proposicional. A continuación vamos a definitionnir la función reduce = fun: Secuente → SistemaG, ésta se encarga de aplicar las reglas de inferencia, en caso de no poder aplicar una regla de inferencia, debido a que no encontró operadores lógicos a la cabeza de la lista, se llama a la función decide. El código es el siguiente:

```
fun reduce(lr(nega(a)::gama,delta)) =
    InRuUno(lr(gama,a::delta)) |
    reduce(lr(conj(a,b)::gama,delta)) =
    InRuUno(lr(a::b::gama,delta)) |
    reduce(lr(gama,nega(a)::delta))   =
    InRuUno(lr(a::gama,delta)) |
```

```
reduce(lr(gama, disy(a,b)::delta)) =
InRuUno(lr(gama, a::b::delta)) |
reduce(lr(gama, impl(a,b)::delta)) =
InRuUno(lr(a::gama, b::delta)) |
reduce(lr(disy(a,b)::gama, delta)) =
InRuDos(lr(a::gama, delta), lr(b::gama, delta)) |
reduce(lr(impl(a,b)::gama, delta)) =
InRuDos(lr(b::gama, delta), lr(gama, a::delta)) |
reduce(lr(gama, conj(a,b)::delta)) =
InRuDos(lr(gama, a::delta), lr(gama, b::delta)) |
reduce(sec)                          = decide(sec);
```

2.4. Ejemplo. La línea

```
reduce (lr(conj(a,b)::gama, delta)) =
InRuUno(lr(a::b::gama, delta))
```

es la codificación de la regla

$$\frac{\Gamma, A, B \rightsquigarrow \Delta}{\Gamma, (A \wedge B) \rightsquigarrow \Delta} \wedge\rightsquigarrow$$

y la línea

```
reduce(lr(impl(a,b)::gama, delta)) =
InRuDos(lr(gama, a::delta), lr(b::gama, delta))
```

es la codificación de la regla

$$\frac{\Gamma \rightsquigarrow A, \Delta \quad \Gamma, B \rightsquigarrow \Delta}{\Gamma, (A \to B) \rightsquigarrow \Delta} \to\rightsquigarrow$$

Como se puede observar una instancia de nuestro único axioma es cualquier secuente $\Gamma \rightsquigarrow \Delta$ tal que $\Gamma \cap \Delta \neq \varnothing$, esto es, contienen alguna proposición en común.

En ML la forma de determinar si una proposición es un teorema será por medio de la función prueba: SistemaG → bool. Esta función se llama recursivamente mientras reciba elementos de tipo InRuUno o InRuDos, es decir las premisas aún contienen conectivos lógicos, con los cuales llama a la función reduce que es la función que aplica las reglas de inferencia. Cuando prueba recibe un elemento de tipo Ax (premisa sin operadores lógicos) entonces llamará a la función intersecta la cual devuelve true si los conjuntos no son disjuntos, es decir la conclusión es una axioma, y false en otro caso.

```
fun prueba(Ax(lr(left, right))) =
    intersecta(left, right) |
    prueba(InRuUno(up))          =
    prueba(printVal(reduce(up))) |
    prueba(InRuDos(up1, up2))    =
    prueba(printVal(reduce(up1))) andalso
    prueba(printVal(reduce(up2)));
```

3. Funciones auxiliares

En esta sección presentaremos funciones auxiliares que son llamadas por las discutidas anteriormente.

3.1. Función intersecta: Secuente → bool. Recordemos que Secuente es de tipo Sec ∗ Sec, es decir tiene una lista del lado izquierdo y otra del lado derecho, así intersecta busca si la parte derecha tiene elementos en común con la parte izquierda, si este es el caso devuelve true, en caso contrario devuelve false. Esto es porque si se tiene una o mas proposiciones iguales del lado derecho y del lado izquierdo se llega a una contradicción, lo cual quiere decir que no hay valores de verdad que hagan falsa a la proposición inicial.

```
fun intersecta([],_)   = false
|   intersecta(h::t,r) = let
                           fun esta(_,[]) = false
                           |   esta(x,h::t) = x = h
                               orelse esta(x,t);
                         in
                           esta(h,r)
                           orelse intersecta(t,r)
                         end;
```

3.2. Función decide: Secuente → SistemaG. La función decide se llama cuando no se ha encontrado un conectivo a la cabeza de gama ni a la cabeza de delta, pero podría haber alguno todavía en el resto de alguna de las dos secuencias, por lo tanto esta función tiene como propósito buscar un conectivo y de hallarlo dejarlo a la cabeza de la secuencia en la que lo halló. Si no lo encuentra en ninguna de las dos secuencias se sospecha de que se trata de una instancia del único axioma.

```
fun decide(lr(gama,delta)) =
        let
          fun hayoperador(nil,pa) = (false,pa)
          |   hayoperador(lp(a)::t,pa) =
              hayoperador(t,lp(a)::pa)
          |   hayoperador(x,pa) = (true,x@pa)
          val (valor,gama1) =
              hayoperador(gama,nil)
        in
          if valor
          then InRuUno(lr(gama1,delta))
          else
            let
              val (valor1,delta1) =
              hayoperador(delta,nil)
            in
              if valor1
              then InRuUno(lr(gama,delta1))
              else Ax(lr(gama,delta))
```

```
        end
    end;
```

Con lo cual tenemos el demostrador completo del sistema de Gentzen.

4. EJEMPLOS

A continuación damos dos ejemplos de como funciona el demostrador y sus correspondientes árboles de demostración.

4.1. EJEMPLO. Para saber si la proposición $((p \to q) \to ((\neg q) \to (\neg p)))$ es siempre verdadera, usando el sistema de Gentzen, tenemos que construir un árbol de deducción para el secuente $\rightsquigarrow ((p \to q) \to ((\neg q) \to (\neg p)))$ o usar el demostrador automático que hemos desarrollado de la siguiente manera:

```
prueba(InRuUno(lr([],[impl(impl(lp("p"),lp("q")),
impl(nega(lp("q")),nega(lp("p"))))])));
```

Obteniendo

```
InRuUno(lr([impl(lp "p", lp "q")],
           [impl(nega(lp "q"), nega(lp "p"))]))
InRuUno(lr([nega(lp "q"), impl(lp "p", lp "q")],
           [nega(lp "p")]))
InRuUno(lr([impl(lp "p", lp "q")],
           [lp "q", nega(lp "p")]))
InRuDos(lr([lp "q"], [lp "q", nega(lp "p")]),
        lr([], [lp "p", lp "q", nega(lp "p")]))
InRuUno(lr([lp "q"],
           [nega(lp "p"), lp "q"]))
InRuUno(lr([lp "p", lp "q"],
           [lp "q"]))
Ax(lr([lp "p", lp "q"],
      [lp "q"]))
InRuUno(lr([],
           [nega(lp "p"), lp "q", lp "p"]))
InRuUno(lr([lp "p"],
           [lp "q", lp "p"]))
Ax(lr([lp "p"],
      [lp "q", lp "p"]))
> val it = true : bool
```

Es decir, la proposición $((p \to q) \to ((\neg q) \to (\neg p)))$ es una tautología. Su árbol de deducción como lo hariamos las personas es el siguiente:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p, q \rightsquigarrow q}\,Ax}{q \rightsquigarrow q, (\neg p)}\,\rightsquigarrow\neg \quad \dfrac{}{\rightsquigarrow p, q, (\neg p)}\,\dfrac{}{p \rightsquigarrow p, q}\,Ax\,\rightsquigarrow\neg}{(p \to q) \rightsquigarrow q, (\neg p)}\,\to\rightsquigarrow}{(\neg q), (p \to q) \rightsquigarrow (\neg p)}\,\neg\rightsquigarrow}{(p \to q) \rightsquigarrow ((\neg q) \to (\neg p))}\,\rightsquigarrow\to}{\rightsquigarrow ((p \to q) \to ((\neg q) \to (\neg p)))}\,\rightsquigarrow\to$$

4.2. Ejemplo. De la misma forma si queremos saber si la proposición $((\neg p) \wedge p)$ es siempre verdadera, podemos construir un árbol de deducción para el secuente $\rightsquigarrow ((\neg p) \wedge p)$ o introducirla al programa de la siguiente manera:

```
prueba(InRuUno(lr([],[conj(nega(lp("p")),lp("p"))])));
```

Obteniendo

```
InRuDos(lr([], [nega(lp "p")]),
        lr([], [lp "p"]))
InRuUno(lr([lp "p"], []))
Ax(lr([lp "p"], []))>
val it = false : bool
```

Es decir, la proposición $((\neg p) \wedge p)$ no es una tautología y se puede falsificar cuando p toma el valor veritativo verdadero (aunque también cuando p es falso, ya que en realidad es una contradicción). Su árbol de deducción es el siguiente:

$$\dfrac{\dfrac{p \rightsquigarrow}{\rightsquigarrow (\neg p)}\rightsquigarrow \neg \qquad \rightsquigarrow p}{\rightsquigarrow ((\neg p) \wedge p)} \rightsquigarrow \wedge$$

Conclusión

Como se ha mostrado ML es muy útil para programar objetos definidos matemáticamente, ya que nos permite modelar fácilmente el alfabeto, la sintaxis y la semántica que se le da a dichos objetos. ML permite tener código elegante y simple, también facilita su entendimiento, interpretación y mantenimiento.

Referencias

[1] Robert Harper; Programming in Standard ML; Carnegie Mellon University, Spring 2005, `http://www.cs.cmu.edu/~rwh/smlbook/offline.pdf`, last visited September 2010.
[2] Jean Gallier; Logic for Computer Science: Foundations of Automatic Theorem Proving; 2003, `http://www.cis.upenn.edu/~jean/gbooks/logic.html`, last visited September 2010.
[3] Steve Reeves and Mike Clarke; Logic for Computer Science, Addison-Wesley Publishers Ltd. 1990, `http://www.cs.waikato.ac.nz/~stever/LCS.pdf`, last visited September 2010.

Facultad de Ciencias de la Computación
Benemérita Universidad Autónoma de Puebla
Av. 14 Sur y Av. San Claudio, CU, San Manuel
Puebla, Puebla, México. jlavalle@cs.buap.mx

Facultad de Ciencias Físico Matemáticas
Benemérita Universidad Autónoma de Puebla
Rio Verde y Av. San Claudio, CU, San Manuel
Puebla, Puebla, México. arrazola@fcfm.buap.mx

Facultad de Ciencias Físico Matemáticas
Benemérita Universidad Autónoma de Puebla
Rio Verde y Av. San Claudio, CU, San Manuel
Puebla, Puebla, México. jp_190999@hotmail.com