TRABAJO FIN DE GRADO

FACULTAD DE MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

LÓGICA COMPUTACIONAL DESDE EL PUNTO DE VISTA DE LA PROGRAMACIÓN FUNCIONAL: ELIMINACIÓN DE CUANTIFICADORES

Realizado por:
María Dolores Mateo Ceballos

Supervisado por:
D. José Antonio Alonso Jiménez
Dª. María José Hidalgo Doblado

Índice general

Abstract

Computational Logic is a wide interdisciplinary field having its theoretical and practical roots in mathematics, computer science, logic, and artificial intelligence. Computational Logic try to produce efficient and powerful algorithms for deciding the satisfiability of formulas in logical theories.

This work is about propositional and first order logic, and its implementation in the functional language Ocaml. In particular, the aim of this work is to explain the quantifier elimination algorithm. As an example, we develop the quantifier elimination algorithm for the Theory of dense linear orders.

Quantifier elimination is an algorithm supported by some logical theories. By eliminating quantifiers from a formula, it makes it possible to test its satisfiability in the sense of propositional logic.

Capítulo 1

Introducción

Según la RAE, "la lógica es la ciencia que expone las leyes, modos y formas de las proposiciones en relación con su verdad o falsedad". Nacida hace más de dos milenios en la Antigua Grecia con Aristóteles y su investigación acerca de los principios del razonamiento válido o correcto, la cual se recoge principalmente en *Órganon*, la lógica ha estado estrechamente ligada al desarrollo intelectual del ser humano, al desarrollo de otras ciencias al establecer las formas correctas de razonamiento y, en particular, al desarrollo de las matemáticas.

La lógica matemática propiamente dicha comienza a desarrollarse principalmente en el siglo XIX, gracias a la contribución de George Boole (1815-1864) y Augustus De Morgan (1806-1871).

El primero es el creador de la conocida Álgebra de Boole, recogida inicialmente en *Análisis Matemático de la Lógica*, publicado en 1847, y extendido en 1854 en *Investigación sobre las Leyes del Pensamiento*.

Augustus De Morgan es conocido, entre otros, por formular las llamadas Leyes de De Morgan. Su obra principal en el campo de la lógica es *Lógica formal*, publicado en 1847.

Entre el final del siglo XIX y el comienzo del siglo XX se sientan las bases de la lógica matemática moderna. Destaca Gottlob Frege (1848-1925), que en su obra *Conceptografía* o *Escritura conceptual* (1879) introduce una nueva sintaxis, en la que destaca la inclusión de los llamados cuantificadores. Las inconsistencias de la teoría puestas de manifiesto en su trabajo motivan a Bertrand Russell y Alfred North Whitehead a publicar, entre 1910 y 1913, un conjunto de tres libros llamado *Principia mathematica*, en un intento de describir un conjunto de axiomas y reglas de inferencia en lógica simbólica a partir de los que se pudieran probar todas las verdades matemáticas. Este intento termina con los teoremas de incompletitud de Gödel, publicados en 1931. En esta época destacan otros grandes matemáticos como Jaques Herbrand o David Hilbert.

Con el desarrollo de los ordenadores a partir de mediados del siglo XX comienza a desarrollarse la lógica computacional, en la que se enmarca este trabajo.

La lógica computacional es un campo interdisciplinar con sus raíces teóricas y prácticas en las matemáticas, la informática, la lógica y la inteligencia artificial.

El objetivo de este trabajo es estudiar conceptos básicos de la lógica proposicional y de primer orden, implemèntandolos en un lenguaje de programación funcional. Esta tarea será desarrollada en los capítulos 3 y 4, basada en el trabajo de J. Harrison [6]. El fin último es estudiar el algoritmo de eliminación de cuantificadores, lo que se hará a lo largo del capítulo 5, y cuya implementación se basa también en el trabajo de J. Harrison. Para el desarrollo teórico nos apoyamos asimismo en los libros de Z. Manna [3], H. B. Enderton [5] y S. M. Srivastava[9].

Para la implementación hemos usado el lenguaje de programación funcional Ocaml [2]. La primera implementación de este lenguaje aparece en 1987 con el nombre de Caml (acrónimo de *Categorical Abstract Machine Language*) y se continúa desarrollando hasta 1992. Es creado por el INRIA (*Institut National de Recherche en Informatique et en Automatique*) en Francia.

Objetive Caml, conocido actualmente como Ocaml, se comienza a desarrollar en 1996, como una mejora de las anteriores versiones Caml Light y Caml Special Light, y se renombra como OCaml en 2011. Este lenguaje ofrece, entre otros:

- Inferencia de tipos, lo cual permite definir operaciones sin explicitar el tipo de los argumentos o del resultado.
- Definiciones de nuevas estructuras de datos y el uso de las definiciones mediante patrones.
- Manejo de errores y excepciones.

Estas facilidades son las que nos han llevado a escoger Ocaml como lenguaje de programación funcional para el desarrollo del presente trabajo.

Éste consta de un primer capítulo en el que introduciremos el lenguaje de programación funcional Ocaml.

A lo largo de los dos siguientes capítulo desarrollaremos la lógica proposicional y de primer orden, implementándola a su vez en Ocaml. Expondremos su sintaxis y semántica, la transformación de fórmulas en diferentes formas normales y algunos procedimientos de decisión. Para finalizar, enunciaremos el Teorema de Herbrand, dejando así constancia de que no existe ningún algoritmo para decidir la satisfacibilidad de una fórmula de primer orden y, por lo tanto, de la importancia de los algoritmos que deciden la satisfacibilidad restringiéndose a conjuntos más pequeños de fórmulas.

Por último, en el capítulo 5, desarrollaremos uno de estos algoritmos para decidir la satisfacibilidad de las fórmulas de una teoría de primer orden. Como ejemplo, veremos un algoritmo para la Teoría de los órdenes lineales densos.

Existen otros trabajos relativos al algoritmo de eliminación de cuantificadores, como el realizado por Amine Chaieb y Tobias Nipkow para la aritmética de Presburger [4]. Nosotros podríamos continuar este trabajo implementando en Ocaml dicho algoritmo para esta teoría, o para otras teorías de primer orden.

Adjuntos a este trabajo se acompañan los archivos correspondientes a la implementación del mismo: “inicio.ml”, “lpsintax.ml”, “lpsem.ml”, “defcnf.ml”, “dp.ml”, “lpo.ml”, “skolem.ml”, “elcuant.ml”.

Capítulo 2

Introducción a Ocaml

En este capítulo introduciremos algunos conceptos del lenguaje de programación funcional Ocaml que se usarán en el desarrollo de este trabajo. Podemos saber más sobre este lenguaje en [8].

En primer lugar debemos instalar Ocaml. Podemos ver cómo hacerlo en [1].

Antes de empezar debemos advertir que:

- Cada instrucción debe acabar con `;;`.
- Los comentarios deben quedar delimitados por `(*` y `*)`.

Los ejemplos de Ocaml que aparecerán a lo largo de este trabajo constan de una primera parte que se inicia con el símbolo # seguido de las instrucciones que introducimos en Ocaml y una segunda parte, la última línea, que nos devolverá Ocaml, cuyo significado explicaremos más adelante.

2.1. Definir expresiones y funciones

La manera más sencilla de definir expresiones y funciones en Ocaml es usar el comando `let`, seguido del nombre de la expresión o la función y sus argumentos, sin usar paréntesis ni comas. Veamos un ejemplo:

```
# let suma a b = a + b;;
val suma : int -> int -> int = <fun>
```

Para llamar a las funciones definidas hay que escribir su nombre y los argumentos a los que se le aplica.

```
# suma 2 3;;
- : int = 5
```

Este comando se usa asimismo para asignarle un valor a una variable. Por ejemplo:

```
# let x = 10;;
val x : int = 10
```

Pero también es posible definir una función anónima, usando la siguiente sintaxis:

`fun` *argumentos* `->` *definición*

Como ya hemos comentado, Ocaml devuelve un mensaje cuando introducimos alguna instrucción. En los ejemplos anteriores devuelve:

```
val suma : int -> int -> int = <fun>

- : int = 5

val x : int = 10
```

En el primer y último caso, en los cuales hemos definido una función o expresión, devuelve su tipo y el de sus argumentos. Esto se debe a que Ocaml usa inferencia de tipos, es decir, no necesita que se indique de qué tipo es cada término (entero, real, cadena, lista...). Sin embargo, Ocaml no hace conversiones automáticas entre tipos. Por ejemplo, si queremos usar números reales debemos escribir `2.0` porque `2` es un entero.

En el segundo caso, hemos evaluado una función para unos datos concretos. Ocaml devuelve el resultado de esta operación.

Si cometemos un error de tipo al definir una función, Ocaml así nos lo avisa:

```
# suma 2.5 3;;
Characters 5-8:
  suma 2.5 3;;
       ^^^
Error: This expression has type float but an expression
was expected of type int
```

A continuación enumeramos algunas formas de definir funciones:

- Definiciones locales:

  `let` *nombre* `=` *expresión1* `in` *expresión2*

  Se define una función como la *expresión2*, en la que aparece la *expresión1*, definida sólo de forma local. Por ejemplo:

  ```
  let operacion a b =
    let c = abs_float (a -. b)
        in sqrt c ;;
  val operacion : float -> float -> float = <fun>
  ```

Se ha definido la función `operacion`, definiendo previamente *c*.

- Condicional `if`:

  `let` *nombre* `=` `if` *condición* `then` *definición1* `else` *definición2*

  Si se verifica la condición se aplica la *definición1* y si no, se aplica la *definición2*. Veamos un ejemplo:

```
let mayor a b = if a > b then a else b;;
val mayor : 'a -> 'a -> 'a = <fun>
```

  Esta función devuelve devuelve el mayor entero entre *a* y *b*.

- Correspondencias de patrones:

  `let` *nombre* `=`
  `match` *objeto* `with`
  patrón1 `->` *definición1*
  `|` *patrón2* `->` *definición2*
  ⋮
  `|` `_` `->` *definiciónn*

  Esto empareja un *objeto* con varios patrones de forma secuencial. Se da una definición de la función para cada emparejamiento. Veamos un ejemplo:

```
let eliminacion l =
  match l with
    [] -> []
  | h::t -> t;;
val eliminacion : 'a list -> 'a list = <fun>
```

  Esta función distingue si la lista tiene algún elemento o si es vacía, eliminando el primer elemento en el primer caso o devolviendo nuevamente la lista vacía en el segundo.

  Notemos que el tipo es `'a list`. Esto significa que los elementos de la lista pueden ser de cualquier tipo `a`.

  Tras usar varios patrones, para englobar todos los casos restantes se puede usar como patrón un guión bajo.

Ocaml además permite que, en la definición de una función, se devuelva un error, y se explicite dicho error. Veamos un ejemplo:

```
let hd l =
  match l with
   h::t -> h
  | _ -> failwith "hd";;
```

Esta función obtiene la cabeza de una lista y, si la lista es vacía, devuelve un error, que definimos mediante la función `failwith`.

Los comandos usados para definir funciones pueden anidarse.

Para definir funciones recursivas se debe explicitar que lo son, usando para definirlas `let rec` en lugar de `let`. Veamos un ejemplo:

```
let rec forall p l =
  match l with
    [] -> true
  | h::t -> p(h) && forall p t;;
val forall : ('a -> bool) -> 'a list -> bool = <fun>
```

Esta función recorre de manera recursiva la lista l verificando si cada uno de sus elementos verifica el predicado p.

2.2. Tipos en Ocaml

Existen muchos tipos predefinidos en este lenguaje, aunque nosotros sólo introduciremos los más comunes:

- Enteros: tipo `int`.
- Reales: tipo `float`.
  Notemos que para distinguir un entero de un real se debe o bien usar el punto decimal o bien especificar un exponente con `e`.
- Caracter: tipo `char`. Implementa los caracteres del código ASCII. Para escribir un elemento de este tipo se utilizan las comillas simples ' '.
- Cadenas: tipo `string`. Una cadena es una secuencia de caracteres. Para escribir un elemento de este tipo se utilizan las comillas .
- Booleanos: tipo `bool`. Solo tienen dos valores: `true` y `false`.
- Tuplas. Es un producto cartesiano de dos o más tipos. Corresponde a una secuencia de valores ordenador por comas. Veamos un ejemplo:

  ```
  # 3,"Carlos",'x';;
  - : int * string * char = (3, "Carlos", 'x')
  ```

- Listas: tipo `'a list`. Una lista es una secuencia de elementos de un mismo tipo, separados por puntos y comas. Las listas se representan entre corchetes.

Además, Ocaml permite definir nuevos tipos de datos. Esto se hace mediante el comando `type`, y la siguiente sintaxis:

`type` *nombre* `=` *constructor-1* `|` ... `|` *constructor-n*

donde el nombre del tipo debe comenzar por una letra minúscula.

Los n constructores pueden ser bien constantes, bien funciones que dependen de otros tipos. En el segundo caso escribiremos cada *constructor-i* como:

nombre-constructor-i `of` *tipo*

donde *tipo* puede ser un sólo tipo o un producto cartesiano de dos o más de ellos.

Veamos un ejemplo:

```
type arboles =
   Vacío
 | Hoja of int
 | Nodo of int * arboles list;;
```

Hemos definido un tipo de datos *arboles*, cuyos constructores son:

- *Vacío*, una constante
- *Hoja*, que toma un valor en los enteros
- *Nodo* que toma como valores un entero y una lista del mismo tipo definido.

2.3. Funciones predefinidas

Se enumeran a continuación algunas funciones predefinidas en Ocaml que usaremos en el desarrollo de este trabajo:

- `+`, `-` y `*` son funciones definidas sobre enteros.
- `not` es la negación, definida sobre los booleanos.
- `&&` y `||` son la conjunción y disyunción, respectivamente, definidas sobre los booleanos.
- `:` es una función definida sobre listas, que añade un elemento a su comienzo.
- `@` es una función definida sobre listas, que las concatena.
- `=`, `<>` son la igualdad y desigualdad, respectivamente. Son funciones polimórficas, es decir, están definidas para diversos tipos.

2.4. Ficheros de archivos

Para cargar un fichero de código se escribe en la terminal:

```
# use "nombre.ml";;
```

Los ficheros que se usan en este trabajo, ordenados por capítulos, deben cargarse en el siquiente orden:

1. Código inicial necesario para este trabajo:
   - `# use " inicio.ml";;`
2. Lógica proposicional:
   - `# use "lpsintax.ml";;`
   - `# use "lpsem.ml";;`
   - `# use "defcnf.ml";;`
   - `# use "dp.ml";;`
3. Lógica de primer orden:
   - `# use "lpo.ml";;`
   - `# use "skolem.ml";;`
4. Eliminación de cuantificadores:
   - `# use " elcuan.ml";;`

Capítulo 3

Lógica proposicional

El objetivo de la lógica es el desarrollo de una lenguaje formal, que carezca de ambigüedad, para modelar enunciados.

La lógica proposicional es la forma más simple de la lógica. Trata sobre la veracidad o falsedad de las proposiciones, esto es, afirmaciones que pueden ser consideradas verdaderas o falsas.

3.1. Sintaxis de la lógica proposicional

Los elementos básicos de la lógica proposicional son:

- los símbolos de verdad $\top$ o 'Verdadero' y $\bot$ o 'Falso', que en nuestra implementación en Ocaml serán `True` y `False`, respectivamente;
- las variable atómicas o proposicionales, denotadas comúnmente 'p', 'q', 'r'...;
- las conectivas lógicas, que pueden tener un argumento, es decir, ser monarias:
  - la negación $\neg$: `Not`,

  o tener dos argumentos, es decir, ser binarias:
  - la conjunción $\wedge$: `And`,
  - la disyunción $\vee$: `Or`,
  - la implicación $\Rightarrow$: `Imp`,
  - la doble implicación, equivalencia o bicondicional $\Leftrightarrow$: `Iff`.

El conjunto de las fórmulas proposicionales se define recursivamente como sigue:

- $\top$ y $\bot$ son fórmulas proposicionales.
- Las variables proposicionales son fórmulas proposicionales.

- Si F y G son fórmulas proposicionales, entonces $\neg F$, $F \vee G$, $F \wedge G$, $F \Rightarrow G$ y $F \Leftrightarrow G$ son fórmulas proposicionales.

Los símbolos de verdad y las variables proposicionales se denominan fórmulas atómicas o átomos.

Para implementar esto en Ocaml definimos un nuevo tipo de dato `formula`:

```
type ('a)formula = False
                 | True
                 | Atom of 'a
                 | Not of ('a)formula
                 | And of ('a)formula * ('a)formula
                 | Or of ('a)formula * ('a)formula
                 | Imp of ('a)formula * ('a)formula
                 | Iff of ('a)formula * ('a)formula
                 | Forall of string * ('a)formula
                 | Exists of string * ('a)formula;;
```

Hemos añadido además los constructores `Forall` y `Exists`, que completan la definición de este tipo en Ocaml pero que no utilizaremos hasta hablar de lógica de primer orden. Por el momento obviaremos ambos constructores en las futuras definiciones de funciones.

Definimos el tipo de las variables proposicionales:

```
type prop = P of string;;
```

y una función para usar el nombre de las proposiciones, como cadenas:

```
let pname(P s) = s;;
```

Con esto podemos implementar las fórmulas proposicionales en Ocaml mediante el tipo `prop formula`.

Para escribir las fórmulas en Ocaml de una manera "agradable" y parecida a la notación seguida en el desarrollo teórico, hemos usado varias funciones de impresión, pero no entraremos en detalles sobre ello.

Establezcamos en lo que sigue una norma de precedencia de los operadores lógicos: negación, conjunción, disyunción, implicación y doble implicación. En caso de igualdad, los asociaremos por la derecha.
Veamos un ejemplo:

```
# let fm = <<p ==> q <=> r /\ s \/ (t <=> ~ ~u /\ v)>>;;
val fm : prop formula = <<p ==> q <=> r /\ s \/ (t <=> ~(~u) /\ v)>>
```

Formalmente, dicha fórmula se escribiría:

$$(p \Rightarrow q) \Leftrightarrow ((r \wedge s) \vee (t \Leftrightarrow (\neg\,(\neg u) \wedge v)))$$

3.1.1. Operaciones sintácticas

Es conveniente tener operaciones sintácticas correspondientes a los constructores de `formula` que podamos usar como funciones de Ocaml.

```
let mk_and p q = And(p,q) and mk_or p q = Or(p,q)
and mk_imp p q = Imp(p,q) and mk_iff p q = Iff(p,q)
and mk_forall x p = Forall(x,p) and mk_exists x p = Exists(x,p);;
```

Al mismo tiempo, dada una fórmula, nos gustaría obtener sus componentes según el operador lógico principal a las que se le aplica. Así tenemos las funciones:

```
let dest_iff fm =
  match fm with Iff(p,q) -> (p,q) | _ -> failwith "dest_iff";;

let dest_and fm =
  match fm with And(p,q) -> (p,q) | _ -> failwith "dest_and";;

let dest_or fm =
  match fm with Or(p,q) -> (p,q) | _ -> failwith "dest_or";;

let dest_imp fm =
  match fm with Imp(p,q) -> (p,q) | _ -> failwith "dest_imp";;
```

De manera similar, las funciones siguientes descomponen una fórmula que contiene conjunciones y disyunciones, devolviendo de manera recursiva una lista formada por las fórmulas proposicionales a las que se aplican dichos operadores:

```
let rec conjuncts fm =
  match fm with And(p,q) -> conjuncts p @ conjuncts q | _ -> [fm];;

let rec disjuncts fm =
  match fm with Or(p,q) -> disjuncts p @ disjuncts q | _ -> [fm];;
```

Para obtener p y q de una fórmula $p \Rightarrow q$, es decir, su antecedente y su consecuente, definimos:

```
let antecedent fm = fst(dest_imp fm);;
let consequent fm = snd(dest_imp fm);;
```

A veces necesitamos definir funciones por recursión sobre fórmulas. La siguiente aplica una función sobre todos los átomos de una fórmula, pero deja la estructura inalterada. Notemos así que Ocaml admite definiciones de funciones de orden superior.

```
let rec onatoms f fm =
  match fm with
    Atom a -> f a
  | Not(p) -> Not(onatoms f p)
  | And(p,q) -> And(onatoms f p,onatoms f q)
  | Or(p,q) -> Or(onatoms f p,onatoms f q)
  | Imp(p,q) -> Imp(onatoms f p,onatoms f q)
  | Iff(p,q) -> Iff(onatoms f p,onatoms f q)
  | Forall(x,p) -> Forall(x,onatoms f p)
  | Exists(x,p) -> Exists(x,onatoms f p)
  | _ -> fm;;
```

La siguiente itera una función binaria a través de todos los átomos de una fórmula.

```
let rec overatoms f fm b =
  match fm with
    Atom(a) -> f a b
  | Not(p) -> overatoms f p b
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) ->
        overatoms f p (overatoms f q b)
  | Forall(x,p) | Exists(x,p) -> overatoms f p b
  | _ -> b;;
```

Una aplicación de esto podría ser obtener todos los átomos de una fórmula o, de manera más general, iterar una función f sobre el conjunto de todos los átomos:

```
let atom_union f fm = setify (overatoms (fun h t -> f(h)@t) fm []);;
```

donde la función `setify` ordena una lista dada y elimina las repeticiones.

3.2. Semántica de la lógica proposicional

Como las fórmulas proposicionales representan afirmaciones que pueden ser verdaderas o falsas, el significado de una fórmula es uno de los dos valores de verdad: 'Verdadero' y 'Falso'. Sin embargo, al igual que una expresión algebraica $x+y+1$ sólo tiene un significado definido cuando conocemos lo que representan las variables x e y, el significado de una fórmula proposicional depende de los valores de verdad asignados a sus átomos. Esto viene dado por una interpretación, que es una aplicación del conjunto de los átomos al conjunto de los valores de verdad 'Verdadero'. 'Falso'.

Para representar dichas interpretaciones en Ocaml usaremos funciones que para cada variable proposicional devuelvan `true` o `false`. Dada una fórmula fm y una interpretación v, la siguiente función evalúa el valor de dicha fórmula fm en la interpretación v:

```
let rec eval fm v =
  match fm with
    False -> false
  | True -> true
  | Atom(x) -> v(x)
  | Not(p) -> not(eval p v)
  | And(p,q) -> (eval p v) && (eval q v)
  | Or(p,q) -> (eval p v) || (eval q v)
  | Imp(p,q) -> not(eval p v) || (eval q v)
  | Iff(p,q) -> (eval p v) = (eval q v);;
```

Notemos que en la definición de esta función no se han considerado los constructores `Forall` y `Exists` de la definición del tipo de dato `formula` pues, como ya comentamos, los obviaremos por el momento.

Incluyamos una tabla de verdad que muestre como el valor de una fórmula viene determinado por el de sus subfórmulas inmediatas.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
Falso	Falso	Verdadero	Falso	Falso	Verdadero	Verdadero
Falso	Verdadero		Falso	Verdadero	Verdadero	Falso
Verdadero	Falso	Falso	Falso	Verdadero	Falso	Falso
Verdadero	Verdadero		Verdadero	Verdadero	Verdadero	Verdadero

Veamos un ejemplo sobre cómo evaluar la fórmula $p \wedge q \Rightarrow q \vee r$, si p, q y r toman los valores 'Verdadero', 'Falso' y 'Verdadero', respectivamente; o 'Verdadero', 'Verdadero' y 'Falso'.

```
# eval <<p /\ q ==> q /\ r>>
     (function P"p" -> true | P"q" -> false | P"r" -> true);;
- : bool = true

# eval <<p /\ q ==> q /\ r>>
     (function P"p" -> true | P"q" -> true | P"r" -> false);;
- : bool = false
```

3.2.1. Escritura de las tablas de verdad

Como indicábamos anteriormente, es posible obtener el conjunto de los átomos de una fórmula, lo cual nos servirá, entre otras cosas, para definir todas las posibles interpretaciones de dicha fórmula.

```
let atoms fm = atom_union (fun a -> [a]) fm;;
```

Además podemos implementar en Ocaml las tablas de verdad para todos los posibles valores de cada átomo. Para ello definimos primero una función `onallvalua-`

`tions` que verifica si otra función `subfn` devuelve `true` sobre cada posible interpretación de los átomos de `ats`, usando una interpretación existente `v` para todos los demás átomos. Cada interpretación es construida redefiniendo `v` sucesivamente para darles a cada átomo los valores `true` y `false`, y llamarlas recursivamente:

```
let rec onallvaluations subfn v ats =
  match ats with
    [] -> subfn v
  | p::ps -> let v' t q = if q = p then t else v(q) in
             onallvaluations subfn (v' false) ps &&
             onallvaluations subfn (v' true) ps;;
```

Ahora podemos usar esta función para escribir tablas de verdad donde aparezca en cada columna el valor de los átomos de la fórmula y de ésta, y por filas todas las posibles interpretaciones.

```
let print_truthtable fm =
  let ats = atoms fm in
  let width = itlist (max ** String.length ** pname) ats 5 + 1 in
  let fixw s = s^String.make(width - String.length s) ' ' in
  let truthstring p = fixw (if p then "true" else "false") in
  let mk_row v =
     let lis = map (fun x -> truthstring(v x)) ats
     and ans = truthstring(eval fm v) in
     print_string(itlist (^) lis ("| "^ans)); print_newline(); true in
  let separator = String.make (width * length ats + 9) '-' in
  print_string(itlist (fun s t -> fixw(pname s) ^ t) ats "| formula");
  print_newline(); print_string separator; print_newline();
  let _ = onallvaluations mk_row (fun x -> false) ats in
  print_string separator; print_newline();;
```

Veamos un ejemplo:

```
# print_truthtable <<p /\ q ==> q /\ r>>;;
p     q     r     | formula
---------------------------
false false false | true
false false true  | true
false true  false | true
false true  true  | true
true  false false | true
true  false true  | true
true  true  false | false
true  true  true  | true
---------------------------
- : unit = ()
```

3.3. Validez, satisfacibilidad y tautologías

Decimos que una interpretación satisface una fórmula F si la evaluación de ésta sobre dicha fórmula devuelve 'Verdadero'. Esta interpretación se llama modelo de F. Para nuestra implementación en Ocaml esto es `eval p v = true`, donde `p` es la fórmula y `v` es la interpretación. Una fórmula se dice que es

- una tautología o una validez lógica si es satisfecha por todas las interpretaciones o, equivalentemente, si el valor de su tabla de verdad es 'Verdadero' en todas las filas.
- satisfacible si existe alguna interpretación que la satisfaga. Esto es, si el valor de alguna de las filas de su tabla de verdad es 'Verdadero'.
- insatisfacible si no existe ninguna interpretación que satisfaga la fórmula o, equivalentemente, si el valor de cada fila de su tabla de verdad es 'Falso'.

Notar además que una fórmula F es insatisfacible si y sólo si $\neg F$ es una tautología.
Para implementar estos conceptos en Ocaml evaluaremos directamente todas las interpretaciones:

```
let tautology fm =
  onallvaluations (eval fm) (fun s -> false) (atoms fm);;
```

Así la función `onallvaluations` recorre todas las posibles interpretaciones de la fórmula `fm` hasta encontrar una que no la satisfaga, devolviendo `false`, o devolviendo `true` si recorre todas las interpretaciones y éstas la satisfacen.
Podemos definir asimismo la satisfacibilidad y la insatisfacibilidad en términos de la función `tautology`.

```
let unsatisfiable fm = tautology(Not fm);;

let satisfiable fm = not(unsatisfiable fm);;
```

A continuación pondremos algunos ejemplos.

```
# tautology <<p \/ ~p>>;;
- : bool = true
# tautology <<p /\ ~p>>;;
- : bool = false
# unsatisfiable <<p /\ ~p>>;;
- : bool = true
# tautology <<p \/ q ==> q \/ (p <=> q)>>;;
- : bool = false
# satisfiable <<p \/ q ==> q \/ (p <=> q)>>;;
- : bool = true
```

3.4. Simplificación y forma normal negativa

En lógica, las formas normales para las fórmulas son de gran importancia, y pueden aportar valiosa información. Las formas normales son fórmulas lógicamente equivalentes a las dadas, con una expresión determinada. Pero antes de proceder a crearlas, es conveniente definir ciertas reglas de simplificación. En primer lugar definiremos algunas reglas de simplificación sobre los operadores lógicos aplicados a los valores de verdad:

- $\neg\top \Leftrightarrow \bot$, $\neg\bot \Leftrightarrow \top$
- $\neg(\neg p) \Leftrightarrow p$
- $p \wedge \bot \Leftrightarrow \bot$, $\bot \wedge p \Leftrightarrow \bot$
- $p \wedge \top \Leftrightarrow p$, $\top \wedge p \Leftrightarrow p$
- $p \vee \bot \Leftrightarrow p$, $\bot \vee p \Leftrightarrow p$
- $p \vee \top \Leftrightarrow \top$, $\top \vee p \Leftrightarrow \top$
- $(\bot \Rightarrow p) \Leftrightarrow \top$, $(p \Rightarrow \top) \Leftrightarrow \top$
- $(\top \Rightarrow p) \Leftrightarrow p$, $(p \Rightarrow \bot) \Leftrightarrow \neg p$
- $(p \Leftrightarrow \top) \Leftrightarrow p$, $(\top \Leftrightarrow p) \Leftrightarrow p$
- $(p \Leftrightarrow \bot) \Leftrightarrow \neg p$, $(\bot \Leftrightarrow p) \Leftrightarrow \neg p$

Podemos implementar en Ocaml dicha simplificación:

```
let psimplify1 fm =
  match fm with
    Not False -> True
  | Not True -> False
  | Not(Not p) -> p
  | And(p,False) | And(False,p) -> False
  | And(p,True) | And(True,p) -> p
  | Or(p,False) | Or(False,p) -> p
  | Or(p,True) | Or(True,p) -> True
  | Imp(False,p) | Imp(p,True) -> True
  | Imp(True,p) -> p
  | Imp(p,False) -> Not p
  | Iff(p,True) | Iff(True,p) -> p
  | Iff(p,False) | Iff(False,p) -> Not p
  | _ -> fm;;
```

que podemos aplicar a una fórmula de manera recursiva:

```
let rec psimplify fm =
  match fm with
  | Not p -> psimplify1 (Not(psimplify p))
  | And(p,q) -> psimplify1 (And(psimplify p,psimplify q))
  | Or(p,q) -> psimplify1 (Or(psimplify p,psimplify q))
  | Imp(p,q) -> psimplify1 (Imp(psimplify p,psimplify q))
  | Iff(p,q) -> psimplify1 (Iff(psimplify p,psimplify q))
  | _ -> fm;;
```

Notemos nuevamente que no hemos considerado los constructores `Forall` y `Exists` de la definición del tipo de dato `formula`.

Veamos un ejemplo de esta simplificación:

```
# psimplify <<(true ==> (x <=> false)) ==> ~(y \/ false /\ z)>>;;
- : prop formula = <<~x ==> ~y>>
```

Un literal es un átomo o su negación. Decimos que un literal es negativo si es la negación de un átomo, y positivo en otro caso. Podemos implementar ambas definiciones en Ocaml suponiendo que las estamos aplicando a un átomo.

```
let negative = function (Not p) -> true | _ -> false;;

let positive lit = not(negative lit);;
```

Estaremos negando un literal si escribimos $\neg p$ si el literal p es positivo, y eliminaremos la negación si es negativo.

```
let negate = function (Not p) -> p | p -> Not p;;
```

Una fórmula está en forma normal negativa o NNF (del inglés, negative normal form) si se construye a partir de literales usando sólo las conectivas binarias $\wedge$ y $\vee$, y la negación aplicada sólo a átomos, además de si es uno de los casos triviales $\bot$ o $\top$.
Podemos transformar una fórmula en otra lógicamente equivalente en forma normal negativa, eliminando los operadores $\Rightarrow$ y $\Leftrightarrow$ en función de las otras conectivas:

$$\begin{array}{lcl} p \Rightarrow q & \Leftrightarrow & \neg\left(p \wedge \neg q\right) \\ p \Leftrightarrow q & \Leftrightarrow & \neg\left(p \wedge \neg q\right) \wedge \neg\left(\neg p \wedge q\right) \end{array}$$

o, lo que es lo mismo,

$$\begin{array}{lcl} p \Rightarrow q & \Leftrightarrow & \neg p \vee q \\ p \Leftrightarrow q & \Leftrightarrow & \neg p \vee q \wedge p \vee \neg q \end{array}$$

y aplicando las leyes de De Morgan y la ley de la doble negación, las cuales enunciamos a continuación:

$$\begin{array}{rcl} \neg(p \wedge q) & \Leftrightarrow & \neg p \vee \neg q \\ \neg(p \vee q) & \Leftrightarrow & \neg p \wedge \neg q \\ \neg\neg p & \Leftrightarrow & p \end{array}$$

Implementamos todo esto en Ocaml para definir la forma normal negativa.

```
let rec nnf fm =
  match fm with
  | And(p,q) -> And(nnf p,nnf q)
  | Or(p,q) -> Or(nnf p,nnf q)
  | Imp(p,q) -> Or(nnf(Not p),nnf q)
  | Iff(p,q) -> Or(And(nnf p,nnf q),And(nnf(Not p),nnf(Not q)))
  | Not(Not p) -> nnf p
  | Not(And(p,q)) -> Or(nnf(Not p),nnf(Not q))
  | Not(Or(p,q)) -> And(nnf(Not p),nnf(Not q))
  | Not(Imp(p,q)) -> And(nnf p,nnf(Not q))
  | Not(Iff(p,q)) -> Or(And(nnf p,nnf(Not q)),And(nnf(Not p),nnf q))
  | _ -> fm;;
```

Aplicamos además la simplificación que definimos anteriormente, y redefinimos la función `nnf`.

```
let nnf fm = nnf(psimplify fm);;
```

Cabe notar que Ocaml permite definir una función usando un nombre ya asignado a otra. Si ambas funciones son aplicadas a argumentos del mismo tipo, como en este caso, la última función sobrescribirá la primera.

Probemos en un ejemplo que la forma normal negativa de una fórmula dada es lógicamente equivalente a dicha fórmula.

```
# let fm = <<(p <=> q) <=> ~(r ==> s)>>;;
val fm : prop formula = <<(p <=> q) <=> ~(r ==> s)>>
# let fm' = nnf fm;;
val fm' : prop formula =
  <<(p /\ q \/ ~p /\ ~q) /\ r /\ ~s \/
    (p /\ ~q \/ ~p /\ q) /\ (~r \/ s)>>
# tautology(Iff(fm,fm'));;
- : bool = true
```

Sin embargo, hemos de destacar que la NNF es significativamente más larga que la fórmula inicial, pues cada bicondicional expande la fórmula como hemos indicado antes. Así pues, podríamos definir una función que mantenga el bicondicional aunque calcule la NNF para el resto de la fórmula.

```
let rec nenf fm =
  match fm with
    Not(Not p) -> nenf p
  | Not(And(p,q)) -> Or(nenf(Not p),nenf(Not q))
  | Not(Or(p,q)) -> And(nenf(Not p),nenf(Not q))
  | Not(Imp(p,q)) -> And(nenf p,nenf(Not q))
  | Not(Iff(p,q)) -> Iff(nenf p,nenf(Not q))
  | And(p,q) -> And(nenf p,nenf q)
  | Or(p,q) -> Or(nenf p,nenf q)
  | Imp(p,q) -> Or(nenf(Not p),nenf q)
  | Iff(p,q) -> Iff(nenf p,nenf q)
  | _ -> fm;;

let nenf fm = nenf(psimplify fm);;
```

3.5. Formas normales disyuntivas y conjuntivas

Se dice que una fórmula está en forma normal disyuntiva o DNF (del inglés, disjunctive normal form) si es una disyunción de conjunciones de literales. Esto es, tiene la siguiente forma:

$$(I_{1,1} \wedge ... \wedge I_{1,m_1}) \vee ... \vee (I_{n,1} \wedge ... \wedge I_{n,m_n})$$

donde los $I_{i,j}$ son literales.

Decimos asimismo que una fórmula está en forma normal conjuntiva o CNF (del inglés, conjunctive normal form) si es una conjunción de disyunciones de literales. Esto es, tiene la siguiente forma:

$$(I_{1,1} \vee ... \vee I_{1,m_1}) \wedge ... \wedge (I_{n,1} \vee ... \vee I_{n,m_n})$$

donde los $I_{i,j}$ son literales.

Ambas son además formas normales negativas, aunque restringen un poco más su definición.

Veamos ahora cómo construir dichas formas normales. En primer lugar, veamos la construcción de las DNF.
Definamos en primer lugar un par de funciones que usaremos a continuación.

```
let list_conj l = if l = [] then True else end_itlist mk_and l;;

let list_disj l = if l = [] then False else end_itlist mk_or l;;
```

Dado un conjunto de literales, representados en una lista, estas funciones devuelven la conjunción y la disyunción de dichos literales, respectivamente. En el caso especial en el que la lista es vacía la función `list_conj` devuelve $\top$ y la función `list_disj` $\bot$. Estas elecciones nos permitirán más adelante poder afirmar que 'todos los literales son verdaderos' en el primer caso, o que 'existe algún literal verdadero' en el segundo.

Basaremos la transformación de una fórmula en forma normal disyuntiva en las siguientes equivalencias (propiedad distributiva):

$$\begin{array}{lcl} p \wedge (q \vee r) & \Leftrightarrow & p \wedge q \vee p \wedge r \\ (p \vee q) \wedge r & \Leftrightarrow & p \wedge r \vee q \wedge r \end{array}$$

Codificamos esto en Ocaml, suponiendo que las subfórmulas inmediatas ya están en DNF:

```
let rec distrib fm =
  match fm with
    And(p,(Or(q,r))) -> Or(distrib(And(p,q)),distrib(And(p,r)))
  | And(Or(p,q),r) -> Or(distrib(And(p,r)),distrib(And(q,r)))
  | _ -> fm;;
```

Ahora, cuando la fórmula de entrada es una conjunción o disyunción, transformamos recursivamente las subfórmulas inmediatas en forma normal disyuntiva. En el caso de la conjunción hacemos uso de la función anterior.

```
let rec rawdnf fm =
  match fm with
    And(p,q) -> distrib(And(rawdnf p,rawdnf q))
  | Or(p,q) -> Or(rawdnf p,rawdnf q)
  | _ -> fm;;
```

Por ejemplo:

```
# rawdnf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
- : prop formula =
<<(p /\ ~p \/ (q /\ r) /\ ~p) \/ p /\ ~r \/ (q /\ r) /\ ~r>>
```

Aunque esta fórmula está en forma normal disyuntiva, algunas disyunciones son completamente redundantes, pues son lógicamente equivalentes a $\bot$, y podríamos omitirlas. Es por este motivo por lo que vamos a definir una DNF simplificada.

Para ello es conveniente representar una fórmula en forma normal disyuntiva como un conjunto de conjuntos de literales. Por ejemplo, podemos representar $(p \wedge q) \vee (p \wedge r)$ como $\{\{p, q\}, \{p, r\}\}$. Como la estructura lógica es siempre una disyunción de conjunciones, y ambas son asociativas y conmutativas, no perdemos nada esencial al hacer dicha transformación, y es fácil volver a la fórmula.
Ahora podemos escribir esta transformación como una función de Ocaml, usando listas.

```
let distrib s1 s2 = setify(allpairs union s1 s2);;

let rec purednf fm =
  match fm with
    And(p,q) -> distrib (purednf p) (purednf q)
  | Or(p,q) -> union (purednf p) (purednf q)
  | _ -> [[fm]];;
```

La estructura esencial es la misma; la función `distrib` simplemente toma dos conjuntos de conjuntos y devuelve la unión de todos los posibles pares de conjuntos procedentes de éstos sin repeticiones. Si lo aplicamos al ejemplo anterior, obtenemos una lista de listas equivalente a la fórmula que proporciona la función `rawdnf` como solución:

```
# purednf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
- : prop formula list list =
[[<<p>>; <<~p>>]; [<<p>>; <<~r>>]; [<<q>>; <<r>>; <<~p>>];
 [<<q>>; <<r>>; <<~r>>]]
```

Pero gracias a la representación de listas es ahora más fácil simplificar la fórmula resultante. Primero definimos una función `trivial` para comprobar si hay literales complementarios de la forma p y $\neg p$ en la misma lista. Hacemos una partición del conjunto de los literales en los positivos y los negativos, y vemos si hay miembros comunes entre los positivos y la negación de los negativos:

```
let trivial lits =
  let pos,neg = partition positive lits in
  intersect pos (image negate neg) <> [];;
```

Ahora podemos filtrar el resultado de `purednf` para eliminar las listas que verifiquen `trivial`. Por ejemplo:

```
# filter (non trivial) (purednf <<(p \/ q /\ r) /\ (~p \/ ~r)>>);;
- : prop formula list list = [[<<p>>; <<~r>>]; [<<q>>; <<r>>; <<~p>>]]
```

Para seguir simplificando la forma normal disyuntiva definamos un nuevo concepto. Sean dos conjuntos de literales C y D. Diremos que C subsume a D si $C \subset D$. Así, si entendemos dichos conjuntos como la conjunción de sus literales, toda interpretación que satisfaga D también debe satisfacer C.
Necesitamos además definir una función que sustituya los átomos por otras fórmulas:

```
let psubst subfn = onatoms (fun p -> tryapplyd subfn p (Atom p));;
```

La siguiente función toma una fórmula en forma normal negativa y devuelve su forma normal disyuntiva con las simplificaciones que hemos explicado, representada como un conjunto de conjuntos de literales.

```
let simpdnf fm =
  if fm = False then [] else if fm = True then [[]] else
  let djs = filter (non trivial) (purednf(nnf fm)) in
  filter (fun d -> not(exists (fun d' -> psubset d' d) djs)) djs;;
```

Notemos que los casos especiales $\bot$ y $\top$ son representados como la lista vacía y la lista conteniendo la lista vacía, respectivamente. Así, si por ejemplo todas las disyunciones de una DNF son contradicciones, la fórmula debe ser lógicamente equivalente a $\bot$; eliminaremos todas y sólo obtendremos la lista vacía. Además, esto es consistente con la interpretación dada para `list_disj` cuando la lista es vacía.
Para volver a la representación como fórmula tras aplicar esta última función definimos:

```
let dnf fm = list_disj(map list_conj (simpdnf fm));;
```

Comprobemos ahora en el ejemplo anterior que la fórmula es equivalente a la definición que hemos dado de forma normal disyuntiva.

```
# let fm = <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
val fm : prop formula = <<(p \/ q /\ r) /\ (~p \/ ~r)>>
# dnf fm;;
- : prop formula = <<p /\ ~r \/ q /\ r /\ ~p>>
# tautology(Iff(fm,dnf fm));;
- : bool = true
```

Ahora, para decidir la satisfacibilidad de una fórmula en forma normal disyuntiva basta ver si alguna de las disyunciones es satisfacible, esto es, si no contiene dos literales complementarios. Pero por nuestra definición simplificada de DNF, no son posibles las contradicciones. Por tanto, siempre que haya alguna disyunción, nuestra fórmula será satisfacible.

Pasemos ahora a transformar una fórmula en forma normal conjuntiva. Aunque vamos a usar nuevamente conjuntos de conjuntos de literales, le damos un nuevo significado: conjunción de disyunciones de literales. Notemos que por las leyes de De Morgan, si:

$$\neg p \Leftrightarrow \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} p_{ij} \quad \text{entonces} \quad p \Leftrightarrow \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} \neg p_{ij}$$

En términos de listas, podemos obtener la forma normal conjuntiva de una fórmula negándola, obteniendo su forma normal disyuntiva y negando los literales que aparecen en ésta.

```
let purecnf fm = image (image negate) (purednf(nnf(Not fm)));;
```

Usamos algunas funciones que ya habíamos definido para implementar DNF. Cuando tenemos literales complementarios en un conjunto, ahora representan una tautología y no una contradicción, y podemos eliminar dicho conjunto por tanto, usando la función `trivial`. Sólo los dos casos especiales `True` y `False` necesitan ser tratados de manera diferente:

```
let simpcnf fm =
  if fm = False then [[]] else if fm = True then [] else
  let cjs = filter (non trivial) (purecnf fm) in
  filter (fun c -> not(exists (fun c' -> psubset c' c) cjs)) cjs;;
```

Y, análogamente al caso anterior, volvemos a la interpretación como fórmula:

```
let cnf fm = list_conj(map list_disj (simpcnf fm));;
```

Por ejemplo:

```
# let fm = <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
val fm : prop formula = <<(p \/ q /\ r) /\ (~p \/ ~r)>>
# cnf fm;;
- : prop formula = <<(p \/ q) /\ (p \/ r) /\ (~p \/ ~r)>>
# tautology(Iff(fm,cnf fm));;
- : bool = true
```

Como vimos anteriormente, podemos verificar rápidamente la satisfacibilidad de una fórmula gracias a su forma normal disyuntiva. Ahora, gracias a su forma normal conjuntiva, podemos decidir su validez. Así, una conjunción $C_1 \wedge ... \wedge C_n$ es válida si cada C_i es válida. Y como cada C_i es una disyunción de literales, es válida precisamente si contiene la disyunción de un literal y su complementario. Gracias a nuestra simplificación, eliminando dicha posibilidad, la fórmula será válida únicamente en el caso de que su CNF simplificada sea $\top$.

3.5.1. Forma normal conjuntiva usando abreviaciones

Aunque ya hemos visto algoritmos para obtener las formas normales conjuntiva y disyuntiva, éstos son poco eficientes en la práctica para fórmulas un poco más complejas. Si relajamos la propiedad de la equivalencia lógica, podemos hacerlo de manera más eficiente.
Dadas dos fórmulas F y F', diremos que son equisatisfacibles si F es satisfacible si y sólo si lo es F'. Se denota $F \approx F'$. Veremos a continuación un algoritmo para transformar una fórmula F en otra en forma normal conjuntiva F' que sea equisatisfacible a la primera.

La idea básica es introducir nuevos átomos como abreviaciones o definiciones para subfórmulas. Veámoslo en un ejemplo. Supongamos dada la siguiente fórmula:

$$(p \vee (q \wedge \neg r)) \wedge s$$

Introducimos un nuevo átomo p_1, no usado antes en la fórmula, para abreviar $q \wedge \neg r$, combinando la fórmula abreviada con la definición de p_1:

$$(p_1 \Leftrightarrow q \wedge \neg r) \wedge (p \vee p_1) \wedge s$$

Procedemos ahora de manera análoga, introduciendo una variable p_2 para abreviar $p \vee p_1$:

$$(p_1 \Leftrightarrow q \wedge \neg r) \wedge (p_2 \Leftrightarrow p \vee p_1) \wedge p_2 \wedge s$$

y p_3 como una abreviación de $p_2 \wedge s$:

$$(p_1 \Leftrightarrow q \wedge \neg r) \wedge (p_2 \Leftrightarrow p \vee p_1) \wedge (p_3 \Leftrightarrow p_2 \wedge s) \wedge p_3$$

Finalmente transformamos cada una de las fórmulas de las conjunciones en forma normal conjuntiva usando los métodos ya vistos:

$$(\neg p_1 \vee q) \wedge (\neg p_1 \vee \neg r) \wedge (p_1 \vee \neg q \vee r) \wedge (\neg p_2 \vee p \vee p_1) \wedge (p_2 \vee \neg p) \wedge (p_2 \vee \neg p_1) \wedge (\neg p_3 \vee p_2) \wedge (\neg p_3 \vee s) \wedge (p_3 \vee \neg p_2 \vee \neg s) \wedge p_3$$

En el peor caso, el coste de transformación de este nuevo algoritmo no es exponencial, lo cual supone una mejora con respecto al que vimos anteriormente que, en el peor caso, era exponencial.

Pasemos a la implementación de este método en Ocaml. Para las nuevas variables proposicionales, usaremos nombres de la forma `p_n`. La siguiente función devuelve un átomo y su índice incrementado en una unidad, listo para usarlo la siguiente vez.

```
let mkprop n = Atom(P("p_"^(string_of_num n))),n +/ Int 1;;
```

donde la función `string_of_num` transforma un elemento de tipo `num` (número) en una cadena.

Por simplicidad, supongamos que las fórmulas de partida han sido pre- simplificadas mediante la función `nenf`; así pues las negaciones sólo son aplicadas a los átomos y las implicaciones han sido eliminadas, aunque no los bicondicionales. La función recursiva principal `maincnf` toma una terna que consiste en la fórmula que queremos transformar, una función con las abreviaciones hechas hasta el momento, y un contador de los índices de las variables. Devuelve una terna similar con la fórmula transformada, las definiciones, añadiendo las nuevas, y un nuevo contador de los índices usados. Descomponemos una fórmula según sus conectivas binarias en las subfórmulas a las que se les aplican; entonces una función `defstep` que hace el trabajo principal las toma como argumentos `op` y `(p,q)`.

```
let rec maincnf (fm,defs,n as trip) =
  match fm with
    And(p,q) -> defstep mk_and (p,q) trip
  | Or(p,q) -> defstep mk_or (p,q) trip
  | Iff(p,q) -> defstep mk_iff (p,q) trip
  | _ -> trip

and defstep op (p,q) (fm,defs,n) =
  let fm1,defs1,n1 = maincnf (p,defs,n) in
  let fm2,defs2,n2 = maincnf (q,defs1,n1) in
  let fm' = op fm1 fm2 in
  try (fst(apply defs2 fm'),defs2,n2) with Failure _ ->
  let v,n3 = mkprop n2 in (v,(fm'|->(v,Iff(v,fm'))) defs2,n3);;
```

Notemos que `maincnf` y `defstep` son mutuamente recurrentes. Dentro de `defstep`, una llamada recursiva a `maincnf` transforma la subfórmula izquierda `p`, devolviendo la fórmula transformada `fm1`, una lista aumentada de definiciones `defs1` y un contador `n1`. La subfórmula derecha `q` junto con la nueva lista de definiciones y el contador son usados en otra llamada recursiva, devolviendo una fórmula transformada `fm2`, nuevas definiciones `defs2` y un contador `n2`. Entonces construimos la fórmula `fm'`, aplicando el constructor `op` que nos proporcionó `maincnf` al principio a las fórmulas `fm1` y `fm2`. A continuación verificamos si ya hay una definición correspondiente a esta fórmula; si es así, devolvemos la variable que define. En otro caso, creamos una nueva variable `v` e insertamos una nueva definición, devolviendo finalmente dicha variable y el nuevo contador después de la llamada a `mkprop`.

Necesitamos estar seguros de que ninguno de nuestros nuevos átomos introducidos ya han aparecido en la fórmula de partida. Esto lo conseguiremos gracias a la función siguiente:

```
let max_varindex pfx =
  let m = String.length pfx in
  fun s n ->
    let l = String.length s in
    if l <= m or String.sub s 0 m <> pfx then n else
    let s' = String.sub s m (l - m) in
    if forall numeric (explode s') then max_num n (num_of_string s')
    else n;;
```

Ahora podemos implementar la función principal. Primero simplificamos la fórmula, obteniendo `fm'`, y usamos esta fórmula para elegir un índice de variable de partida apropiado, añadiendo 1 al mayor n para el que ya existe una variable `p_n`. Entonces llamamos a la función principal, que mantenemos como un parámetro `fn` para posibles modificaciones futuras, empezando sin abreviaciones o definiciones

y con el índice inicial como contador. Devolvemos la forma normal conjuntiva resultante representada como conjunto de conjuntos:

```
let mk_defcnf fn fm =
  let fm' = nenf fm in
  let n = Int 1 +/ overatoms (max_varindex "p_" ** pname) fm' (Int 0) in
  let (fm'',defs,_) = fn (fm',undefined,n) in
  let deflist = map (snd ** snd) (graph defs) in
  unions(simpcnf fm'' :: map simpcnf deflist);;
```

Por último, para transformar la lista de listas que devuelve `mk_defncf` en la fórmula equivalente, se define:

```
let defcnf fm = list_conj(map list_disj(mk_defcnf maincnf fm));;
```

Veámoslo en el ejemplo que usamos al principio para explicar la transformación:

```
# defcnf <<(p \/ (q /\ ~r)) /\ s>>;;
- : prop formula =
<<(p \/ p_1 \/ ~p_2) /\
  (p_1 \/ r \/ ~q) /\
  (p_2 \/ ~p) /\
  (p_2 \/ ~p_1) /\
  (p_2 \/ ~p_3) /\
  p_3 /\
  (p_3 \/ ~p_2 \/ ~s) /\ (q \/ ~p_1) /\ (s \/ ~p_3) /\ (~p_1 \/ ~r)>>
```

Sin embargo, podemos optimizar el procedimiento evitando algunas definiciones redundantes. En primer lugar, cuando la fórmula inicial se trata de conjunciones iteradas, podemos pasar cada subfórmula inmediata a forma normal conjuntiva separadamente y unirlas después. Si además estas subfórmulas contienen disyunciones podemos seguir descendiendo a través de ellas sin introducir nuevas definiciones o abreviaciones.

Para codificarlo, primero descendemos a través de conjunciones y disyunciones anidadas, antes de empezar a introducir variables de definición. La siguiente función `subcnf` tiene la misma estructura que `defstep` pero no introduce nuevas definiciones, y tiene un parámetro adicional `sfn`:

```
let subcnf sfn op (p,q) (fm,defs,n) =
  let fm1,defs1,n1 = sfn(p,defs,n) in
  let fm2,defs2,n2 = sfn(q,defs1,n1) in (op fm1 fm2,defs2,n2);;
```

Usamos la función anterior primero para definir una función que descienda recursivamente a través de las disyunciones para transformar las subfórmulas inmediatas:

```
let rec orcnf (fm,defs,n as trip) =
  match fm with
    Or(p,q) -> subcnf orcnf mk_or (p,q) trip
  | _ -> maincnf trip;;
```

y a su vez para definir una función que descienda recursivamente a través de las conjunciones llamando a la función `orcnf`:

```
let rec andcnf (fm,defs,n as trip) =
  match fm with
    And(p,q) -> subcnf andcnf mk_and (p,q) trip
  | _ -> orcnf trip;;
```

Ahora la función principal es la misma excepto que se usa `andcnf` en lugar de `maincnf`. Definimos a continuación dos funciones: la primera devuelve el resultado representado como una lista de listas, mientras que la segunda lo hace representado como una fórmula:

```
let defcnfs fm = mk_defcnf andcnf fm;;

let defcnf fm = list_conj (map list_disj (defcnfs fm));;
```

Veamos un ejemplo:

```
# defcnf <<(p \/ (q /\ ~r)) /\ s>>;;
- : prop formula =
<<(p \/ p_1) /\ (p_1 \/ r \/ ~q) /\ (q \/ ~p_1) /\ s /\ (~p_1 \/ ~r)>>
```

3.6. El procedimiento Davis-Putnam

El procedimiento Davis-Putnam es un método para decidir la satisfacibilidad de una fórmula proposicional en forma clausal. Actualmente hay dos algoritmos diferentes, llamados comúnmente 'Davis-Putnam'. El algoritmo original se denomina 'Davis-Putnam' (DP), y el segundo algoritmo, que es una variante de éste presentada más tarde, se denomina 'Davis-Putnam-Loveland-Logemann' (DPLL). Hablaremos primero del algoritmo DP.

3.6.1. Lógica de cláusulas

Una cláusula es un conjunto finito de literales $\{L_1, ..., L_n\}$. Lo denotaremos generalmente por C. Denotaremos a los conjuntos de cláusulas por S.

Diremos que una interpretación I es modelo de una cláusula C si es modelo de alguno de sus literales. Lo notaremos $I \models C$. Por tanto $C = \{L_1, ..., L_n\}$ es

equivalente a la fórmula $F = L_1 \vee ... \vee L_n$, pues $I(C) = I(F)$ para cualquier interpretación I.

Diremos que una interpretación I es modelo de un conjunto de cláusulas S si es modelo de todas sus cláusulas. Lo notaremos $I \models S$. Por tanto:

$$S = \{\{L_{1,1}, ..., L_{1,k_1}\}, ..., \{L_{m,1}, ..., L_{m,k_m}\}\}$$

es equivalente a la fórmula $F = (L_{1,1} \vee ... \vee L_{1,k_1}) \wedge ... \wedge (L_{m,1} \vee ... \vee L_{m,k_m})$.

Ya hemos visto que toda fórmula se puede escribir como una conjunción de disyunciones, es decir, en forma normal conjuntiva. Por tanto, toda fórmula F es equivalente a un conjunto de cláusulas, lo cual se denomina la forma de clausal de F.

Un conjunto de cláusulas es consistente si tiene algún modelo, e inconsistente si no lo tiene. La cláusula vacía, denotada $\square$, no tiene modelos. Por tanto, todo conjunto conteniendo la cláusula vacía es inconsistente. Sin embargo, como toda interpretación es modelo del conjunto vacío de cláusulas, éste es consistente.
En Ocaml representaremos las cláusulas por listas y los conjuntos de cláusulas como listas de listas. Los casos triviales, es decir, el conjunto vacío de cláusulas y el conjunto conteniendo la cláusula vacía serán representados por la lista vacía y la lista conteniendo una lista vacía, respectivamente.

El procedimiento Davis-Putnam se aplica a un conjunto de cláusulas S, transformándolo hasta obtener:

- $\square \in S$, en cuyo caso S es inconsistente.
- $S = \emptyset$, en cuyo caso S es consistente.

En el primer caso la fórmula cuya forma clausal es S es insatisfacible, mientras que en el segundo es satisfacible. Hay tres transformaciones básicas que preservan la satisfacibilidad usadas en el procedimiento DP:

- Regla de eliminación unitaria.
- Regla de eliminación de literales puros.
- Regla de resolución proposicional.

Las dos primeras reglas hacen el conjunto de cláusulas más simple, reduciendo el número total de literales. Por ello aplicamos estas reglas tanto como nos sea posible. La tercera regla es aplicada sólo cuando no se pueden aplicar las anteriores, pues incrementa el tamaño de la forma clausal.

Regla de eliminación unitaria.
Esta regla puede ser aplicada si existe una cláusula unidad, esto es, una cláusula conteniendo un único literal. En ese caso, dicho literal debe ser cierto para que lo sea el conjunto de cláusulas. Esta regla consiste en eliminar todas las cláusulas que

contienen el literal, incluyendo la cláusula unidad, y eliminar de todas las demás cláusulas la negación de dicho literal.
Para implementarlo como una lista de listas, una cláusula unidad será una lista de longitud 1.

```
let one_literal_rule clauses =
  let u = hd (find (fun cl -> length cl = 1) clauses) in
  let u' = negate u in
  let clauses1 = filter (fun cl -> not (mem u cl)) clauses in
  image (fun cl -> subtract cl [u']) clauses1;;
```

Si no hay ninguna cláusula unidad, dicha función devolverá una excepción. Esto hace fácil aplicarla repetidamente hasta que no haya más cláusulas unidad.

Regla de eliminación de literales puros.
Esta regla se basa en el hecho de que si un literal ocurre sólo positivamente o sólo negativamente en el conjunto de todas las cláusulas, podemos eliminar dichas cláusulas preservando la satisfacibilidad. A estos literales los llamamos literales puros. Para implementar esta regla comenzamos tomando el conjunto de todos los literales que aparecen en el conjunto de cláusulas. Hacemos una partición de dicho conjunto en los literales positivos y negativos, obteniendo a continuación los literales puros, y eliminando todas las cláusulas que contienen alguno de ellos.

```
let affirmative_negative_rule clauses =
  let neg',pos = partition negative (unions clauses) in
  let neg = image negate neg' in
  let pos_only = subtract pos neg and neg_only = subtract neg pos in
  let pure = union pos_only (image negate neg_only) in
  if pure = [] then failwith "affirmative_negative_rule" else
    filter (fun cl -> intersect cl pure = []) clauses;;
```

Nuevamente esta función devuelve un fallo si no existen literales puros.

Regla de resolución proposicional.
Esta regla es la única que incrementa el tamaño de la fórmula. Sin embargo, elimina completamente cualquier átomo sin ningún requerimiento especial de las cláusulas que lo contienen. Se aplica cuando un literal ocurre positivamente en alguna cláusula y negativamente en otra distinta. Tengamos en cuenta que si hemos eliminado previamente las tautologías y los literales puros, cualquier literal tendrá esta propiedad.

Sea S un conjunto de cláusulas, y p un literal en las condiciones anteriores. Podemos escribir S como $S = \{\{p\} \cup C_i | 1 \leq i \leq m\} \cup \{\{\neg p\} \cup D_j | 1 \leq j \leq n\} \cup S_0$, donde C_i y D_j son conjuntos de literales en los que no aparecen p, $\neg p$; y S_0 es un conjunto de cláusulas para las que ocurre lo mismo.

Sea I una interpretación tal que $I \models S$. Así, si p es cierto en I también deben de serlo todos los D_j, y sí lo es $\neg p$ lo serán a su vez todos los C_i. En cualquiera

caso, $C_i \cup D_j$ se verificará en dicha interpretación. Por tanto, podemos transformar S en el conjunto de cláusulas $S' = \{C_i \cup D_j | 1 \leq i \leq m, 1 \leq j \leq n\} \cup S_0$.

Recíprocamente, sea I' una interpretación tal que $I' \models S'$. Si existe un C_k que no sea cierto en ella, deben ser ciertos todos los D_j, pues $C_k \cup D_j$ sí debe verificarse, y podríamos tomar p verdadero en dicha interpretación, para la que S se satisfaría. El razonamiento es análogo si algún D_l no es cierto. Por tanto, esta regla mantiene la equisatisfacibilidad.

Llamaremos a la cláusula $C_i \cup D_j$ la resolvente de $\{p\} \cup C_i$ y $\{\neg p\} \cup D_j$, y diremos que se ha obtenido por resolución o, más concretamente, por resolución en p. Implementemos este proceso en Ocaml:

```
let resolve_on p clauses =
  let p' = negate p and pos,notpos = partition (mem p) clauses in
  let neg,other = partition (mem p') notpos in
  let pos' = image (filter (fun l -> l <> p)) pos
  and neg' = image (filter (fun l -> l <> p')) neg in
  let res0 = allpairs union pos' neg' in
  union other (filter (non trivial) res0);;
```

Para aplicar esta regla, debemos decidir respecto de qué literal vamos a hacer la resolución. Dado un literal l, podemos predecir el cambio en el número de cláusulas resultantes de la resolución en l:

```
let resolution_blowup cls l =
  let m = length(filter (mem l) cls)
  and n = length(filter (mem (negate l)) cls) in
  m * n - m - n;;
```

Resolveremos respecto del literal que minimice esta función.

```
let resolution_rule clauses =
  let pvs = filter positive (unions clauses) in
  let p = minimize (resolution_blowup clauses) pvs in
  resolve_on p clauses;;
```

3.6.2. Procedimiento DP

Definiremos este proceso de manera recursiva. Intentaremos aplicar sucesivamente y en este orden las reglas de eliminación unitaria, de eliminación de literales puros y de resolución proposicional con cada nuevo conjunto de cláusulas que obtengamos, hasta obtener la lista vacía, en cuyo caso devolveremos `true`, o la lista conteniendo la lista vacía, devolviendo `false`. Esta recursión debe terminar, pues con cada regla reducimos el número de átomos distintos y, con las dos primeras, también el de cláusulas.

```
let rec dp clauses =
  if clauses = [] then true else if mem [] clauses then false else
  try dp (one_literal_rule clauses) with Failure _ ->
  try dp (affirmative_negative_rule clauses) with Failure _ ->
    dp(resolution_rule clauses);;
```

Podemos usar este procedimiento para determinar la satisfacibilidad de una fórmula. También podemos determinar su validez a través de la negación de dicha fórmula.

```
let dpsat fm = dp(defcnfs fm);;

let dptaut fm = not(dpsat(Not fm));;
```

Por ejemplo:

```
# tautology <<(p \/ (q /\ ~r)) /\ s>>;;
- : bool = false
# dptaut <<(p \/ (q /\ ~r)) /\ s>>;;
- : bool = false
```

3.6.3. Procedimiento DPLL

Para problemas más difíciles, el número y tamaño de las cláusulas generadas en el procedimiento DP puede crecer enormemente. Es esto lo que motivó a Davis, Logemann y Loveland a reemplazar la regla de resolución por una regla de división. Si ninguna de las dos primeras reglas vistas anteriormente son aplicables, entonces podemos elegir un literal p, y la consistencia de un conjunto de cláusulas S se reduce a decidir la consistencia de $S \cup \{p\}$ o de $S \cup \{\neg p\}$. Así, si $S \cup \{p\}$ es consistente, existiría una interpretación en la que p sería verdadero y que también sería modelo de S. Si fuera consistente $S \cup \{\neg p\}$, debería existir una interpretación que fuera modelo de S y en la que p fuera falso. Tras aplicar esta regla, podríamos aplicar seguidamente la regla de eliminación unitaria, que reduciría el conjunto de cláusulas. Es por ello que tenemos garantizada la finalización de este procedimiento.

Análogamente al procedimiento DP, debemos elegir con qué literal aplicaremos la regla de división. Parece sensato escoger aquel que aparece más veces, tanto positiva como negativamente, pues la posterior aplicación de la regla de eliminación unitaria provocaría una mayor simplificación. Definimos así un contador del número de veces que aparece cada literal en un conjunto de cláusulas:

```
let posneg_count cls l =
  let m = length(filter (mem l) cls)
  and n = length(filter (mem (negate l)) cls) in
  m + n;;
```

Ahora podemos definir un algoritmo análogo al DP, pero reemplazando la regla de resolución proposicional por la regla de división.

```
let rec dpll clauses =
  if clauses = [] then true else if mem [] clauses then false else
  try dpll(one_literal_rule clauses) with Failure _ ->
  try dpll(affirmative_negative_rule clauses) with Failure _ ->
  let pvs = filter positive (unions clauses) in
  let p = maximize (posneg_count clauses) pvs in
  dpll (insert [p] clauses) or dpll (insert [negate p] clauses);;
```

Como podemos ver en este algoritmo, la regla de división genera un árbol tal que si todas sus ramas contienen a $\square$, entonces el conjunto inicial de cláusulas es inconsistente; mientras que si alguna rama es el $\emptyset$, entonces el conjunto inicial de cláusulas es consistente.

Una vez más, podemos aplicarlo a la verificación de la satisfacibilidad o validez.

```
let dpllsat fm = dpll(defcnfs fm);;

let dplltaut fm = not(dpllsat(Not fm));;
```

Y, como ejemplo:

```
# dplltaut <<(p \/ (q /\ ~r)) /\ s>>;;
- : bool = false
```

Capítulo 4

Lógica de primer orden

La lógica proposicional sólo nos permite construir fórmulas a partir de proposiciones primitivas que pueden ser independientemente verdaderas o falsas. Sin embargo, esto es demasiado restrictivo para captar los patrones de razonamiento donde la verdad o falsedad de las proposiciones dependen de los valores de las variables no proposicionales.

La lógica de primer orden extiende la lógica proposicional con variables, predicados, funciones y cuantificadores.

En este capítulo consideraremos la lógica de primer orden sin igualdad.

4.1. Sintaxis de la lógica de primer orden

Un lenguaje de primer orden Σ es un conjunto de funciones, predicados y símbolos de constantes, a partir de los cuales se construyen términos y fórmulas, pudiendo usar también para ello variables.

Un lenguaje de primer orden está formado por:

- Símbolos lógicos:
  - las variables, denotadas por $x, y, z, ...$;
  - las conectivas lógicas heredadas de la lógica proposicional;
  - y los cuantificadores, de los que hablaremos posteriormente.
- Símbolos propios:
  - símbolos de función, denotados por $f, g, ...$;
  - símbolos de predicado, $P, Q, R, ...$;
  - símbolos de constantes.

Las funciones y los predicados se aplican a un número determinado de argumentos, llamado aridad. Las constantes pueden considerarse funciones de aridad cero y las variables proposicionales, predicados de aridad cero.

Los términos denotan 'objetos' (números, personas...) y se construyen a partir de variables, constantes y funciones aplicadas a otros términos.

```
type term = Var of string
          | Fn of string * term list;;
```

Como consideramos las constantes funciones de aridad cero, este caso estaría recogido en el constructor `Fn` del tipo `term`.

Se definen las fórmulas atómicas como un predicado cuyos argumentos son términos. En el lenguaje de primer orden sin igualdad se construyen las fórmulas recursivamente como sigue:

- Fórmulas atómicas: $P(t_1, ..., t_n)$, donde P es un símbolo de relación n-aria y $t_1, ..., t_n$ son términos.
- Aplicación de conectivas lógicas a fórmulas: $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$, $F \Leftrightarrow G$, donde F y G son fórmulas.
- Aplicación de cuantificadores a fórmulas: $\forall x.F$, $\exists x.F$, donde F son fórmulas.

Creamos un nuevo tipo `fol` de fórmulas atómicas de primer orden:

```
type fol = R of string * term list;;
```

En este tipo `fol` la cadena representa el símbolo de relación y la lista de términos, los argumentos a los que se aplica dicho símbolo de relación.

Mediante el tipo `fol formula` se representan las fórmulas de primer orden.

Como ya habíamos introducido, la lógica de primer orden está formada, entre otros, por cuantificadores. Estos cuantificadores son:

- El cuantificador universal, $\forall$. La fórmula $\forall x.p$ intuitivamente significa 'para todos los valores de x, p es verdadero'. Lo implementamos en Ocaml como `Forall("x",p)`.
- El cuantificador existencial, $\exists$. La fórmula $\exists x.p$ intuitivamente significa 'p es verdadero para algún o algunos valores de x'. Lo implementamos en Ocaml como `Exists("x",p)`.

En lógica de primer orden los cuantificadores sólo pueden cuantificar variables, no funciones o predicados. Denominaremos alcance del cuantificador a la subfórmula a la que se aplica.

Una aparición de la variable x en la fórmula F se denomina ligada si es en una subfórmula de F de la forma $\forall xG$ o $\exists xG$. En otro caso diremos que es libre. Una variable es libre en una fórmula si tiene alguna aparición libre en ella, y es ligada si tiene alguna aparición ligada. Cabe destacar que una variable puede ser libre y ligada al mismo tiempo, por ejemplo, en la fórmula $R(x,a) \wedge \forall x.R(y,x)$ la variable x es tanto libre como ligada.

Las siguientes funciones devuelven, respectivamente, el conjunto de todas las variables que intervienen en un término y de todas las variables que ocurren en una fórmula:

```
let rec fvt tm =
  match tm with
    Var x -> [x]
  | Fn(f,args) -> unions (map fvt args);;

let rec var fm =
   match fm with
    False | True -> []
  | Atom(R(p,args)) -> unions (map fvt args)
  | Not(p) -> var p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> union (var p) (var q)
  | Forall(x,p) | Exists(x,p) -> insert x (var p);;
```

También podemos obtener el conjunto de todas las variables libres que aparecen en una fórmula:

```
let rec fv fm =
  match fm with
    False | True -> []
  | Atom(R(p,args)) -> unions (map fvt args)
  | Not(p) -> fv p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> union (fv p) (fv q)
  | Forall(x,p) | Exists(x,p) -> subtract (fv p) [x];;
```

Se dice que una fórmula es cerrada o una sentencia si no tiene variables libres, y abierta en caso contrario.

En cuanto a la precedencia de asociación de conectivas y cuantificadores, tomaremos en lo que sigue la siguiente convención: el alcance de los cuantificadores se extenderá tanto a la derecha como sea posible, por ejemplo, $\forall x.P(x) \Rightarrow Q(x)$ significa $\forall x.(P(x) \Rightarrow Q(x))$ y no $(\forall x.P(x)) \Rightarrow Q(x)$. Además, cuando apliquemos

un cuantificador a varias variables, sólo escribiremos el símbolo del cuantificador una vez. El orden de una secuencia de cuantificadores del mismo tipo no debe importar, pero si se trata de cuantificadores de distinto tipo, sí que importa.

Usamos funciones para que la escritura de las fórmulas sea 'agradable'.

Veamos algunos ejemplos:

```
# <<(forall x. x < 2 ==> 2 * x <= 3) \/ false>>;;
- : fol formula = <<(forall x. x < 2 ==> 2 * x <= 3) \/ false>>

# <<forall x y. exists z. x < z /\ y < z>>;;
- : fol formula = <<forall x y. exists z. x < z /\ y < z>>

# <<~(forall x. P(x)) <=> exists y. ~P(y)>>;;
- : fol formula = <<~(forall x. P(x)) <=> (exists y. ~P(y))>>
```

4.2. Semántica de la lógica de primer orden

Como en lógica proposicional, el significado de una fórmula de primer orden es definido recursivamente y depende de los significados dados a sus componentes. En lógica proposicional estas componentes son sólo variables proposicionales, pero en lógica de primer orden tanto las constantes como los símbolos de función y predicado necesitan ser interpretados.

En primer lugar necesitamos definir un dominio o universo del lenguaje $D \neq \emptyset$ donde interpretar todos los términos. Puede ser finito o infinito. Definimos a continuación una estructura del lenguaje como un par $\mathcal{I} = (D, I)$ donde I es una función de interpretación que dota de significado a cada elemento del lenguaje:

- para cada constante c (símbolo de función 0-aria), $I(c) \in D$;
- para cada símbolo de función n-aria f, $I(f)$ es una aplicación $f_{\mathcal{I}} : D^n \longrightarrow D$;
- para cada variable proposicional p (símbolo de relación 0-ario), $I(p) \in \{0, 1\}$;
- para cada símbolo de predicado n-ario P ($n > 0$), $I(P)$ es una aplicación $P_{\mathcal{I}} : D^n \longrightarrow \{\text{Verdadero, Falso}\}$. Equivalentemente podemos verlo como un subconjunto $P_M \subseteq D^n$.

Una asignación A en una estructura (D, I) es una función $A : Var \longrightarrow D$ que hace corresponder a cada variable un elemento del universo de la estructura.

Una interpretación del lenguaje es un par $(\mathcal{I}, A)$ formado por una estructura $\mathcal{I}$ del lenguaje y una asignación A en $\mathcal{I}$.

Se define el valor de un término tm en una estructura $\mathcal{I}$ respecto de una asignación A como una función:

$$\mathcal{I}_A(tm) = \begin{cases} I(c) & \text{, si } tm \text{ es una constante c;} \\ A(x) & \text{, si } tm \text{ es una variable x;} \\ I(f)(\mathcal{I}_A(t_1), ..., \mathcal{I}_A(t_n)) & \text{, si } tm \text{ es } f(t_1, ..., t_n) \end{cases}$$

Lo implementamos en OCaml a través de la siguiente función `termval` definida por recursión. En ella,

- `domain` representa el dominio D,
- `func` representa la función de interpretación $f_{\mathcal{I}}$,
- `pred` representa la función de interpretación $P_{\mathcal{I}}$,
- `v` es una función que representa una asignación,
- `tm` es un término.

```
let rec termval (domain,func,pred as m) v tm =
  match tm with
    Var(x) -> apply v x
  | Fn(f,args) -> func f (map (termval m v) args);;
```

Hagamos dos observaciones:

- Mediante `as` se denota la terna `domain,func,pred` por `m`.
- Aunque en la definición de término hemos distinguido tres casos, las constantes son funciones de aridad 0. Por ello solo distinguimos dos casos en la implementación en Ocaml de la función anterior.

Se define el valor de una fórmula F en una estructura $\mathcal{I}$ respecto de una asignación A como una función $\mathcal{I}_A$ tal que:

- Si F es $P(t_1, ..., t_n)$, entonces $\mathcal{I}_A(F) = P_{\mathcal{I}}(\mathcal{I}_A(t_1), ..., \mathcal{I}_A(t_n))$
- Si F es $\neg G$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Verdadero} & \text{si } \mathcal{I}_A(G) = \text{ Falso} \\ \text{Falso} & \text{si } \mathcal{I}_A(G) = \text{ Verdadero} \end{cases}$
- Si F es $G \wedge H$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Verdadero} & \text{si } \mathcal{I}_A(G) = \mathcal{I}_A(H) = \\ & = \text{ Verdadero} \\ \text{Falso} & \text{e.o.c} \end{cases}$
- Si F es $G \vee H$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Falso} & \text{si } \mathcal{I}_A(G) = \mathcal{I}_A(H) = \text{Falso} \\ \text{Verdadero} & \text{e.o.c} \end{cases}$
- Si F es $G \Rightarrow H$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Falso} & \text{si } \mathcal{I}_A(G) = \text{ Verdadero} \\ & \text{y } \mathcal{I}_A(H) = \text{ Falso} \\ \text{Verdadero} & \text{e.o.c} \end{cases}$

- Si F es $G \Leftrightarrow H$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Verdadero} & \text{si } \mathcal{I}_A(G) = \mathcal{I}_A(H) \\ \text{Falso} & \text{e.o.c} \end{cases}$
- Si F es $\forall x.G$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Verdadero} & \text{si para todo } u \in D \text{ se tiene} \\ & \mathcal{I}_{A[x/u]}(G) = \text{ Verdadero} \\ \text{Falso} & \text{e.o.c} \end{cases}$
- Si F es $\exists x.G$, entonces $\mathcal{I}_A(F) = \begin{cases} \text{Verdadero} & \text{si existe algún } u \in D \text{ tal que} \\ & \mathcal{I}_{A[x/u]}(G) = \text{ Verdadero} \\ \text{Falso} & \text{e.o.c} \end{cases}$

donde $A\,[x/u]\,(y) = \begin{cases} u & \text{si } y = x \\ A(y) & \text{si } y \neq x \end{cases}$

Lo implementamos en OCaml a través de la siguiente función definida por recursión, donde, de nuevo:

- `domain` representa el dominio D,
- `func` representa la función de interpretación $f_{\mathcal{I}}$,
- `pred` representa la función de interpretación $P_{\mathcal{I}}$,
- `v` es una función que representa una asignación,
- `fm` es una fórmula.

```
let rec holds (domain,func,pred as m) v fm =
  match fm with
    False -> false
  | True -> true
  | Atom(R(r,args)) -> pred r (map (termval m v) args)
  | Not(p) -> not(holds m v p)
  | And(p,q) -> (holds m v p) & (holds m v q)
  | Or(p,q) -> (holds m v p) or (holds m v q)
  | Imp(p,q) -> not(holds m v p) or (holds m v q)
  | Iff(p,q) -> (holds m v p = holds m v q)
  | Forall(x,p) -> forall (fun a -> holds m ((x |-> a) v) p) domain
  | Exists(x,p) -> exists (fun a -> holds m ((x |-> a) v) p) domain;;
```

Se tiene una propiedad interesante. Si dos asignaciones v y v' coinciden sobre todas las variables libres de una fórmula F, entonces

$$\texttt{holds}\ M\ v\ F = \texttt{holds}\ M\ v'\ F.$$

Para clarificar estos conceptos, veamos algún ejemplo sobre la interpretación de fórmulas en las que intervienen los símbolos de constantes 0 y 1, los símbolos de funciones binarias + y $\cdot$ y el símbolo de predicado binario =. Consideremos una interpretación en la que + sea una 'disyunción exclusiva':

```
let bool_interp =
  let func f args =
    match (f,args) with
      ("0",[]) -> false
    | ("1",[]) -> true
    | ("+",[x;y]) -> not(x = y)
    | ("*",[x;y]) -> x & y
    | _ -> failwith "uninterpreted function"
  and pred p args =
    match (p,args) with
      ("=",[x;y]) -> x = y
    | _ -> failwith "uninterpreted predicate" in
  ([false; true],func,pred);;
```

Si todas las variables son ligadas en una fórmula, la asignación no juega ningún papel en la interpretación de dicha fórmula. Usaremos `undefined` como asignación.

Veamos el valor de las fórmulas

$$\forall x.((x = 0) \vee (x = 1))$$

y

$$\forall x.(x \neq 0 \Rightarrow \exists y.\ x * y = 1)$$

en la interpretación anterior:

```
# holds bool_interp undefined <<forall x. (x = 0) \/ (x = 1)>>;;
- : bool = true

# let fm = <<forall x. ~(x = 0) ==> exists y. x * y = 1>>;;
val fm : fol formula = <<forall x. ~x = 0 ==> (exists y. x * y = 1)>>
# holds bool_interp undefined fm;;
- : bool = true
```

4.2.1. Validez y satisfacibilidad

Una interpretación $(\mathcal{I}, A)$ del lenguaje se dice que es una realización de una fórmula F si $\mathcal{I}_A(F) =$ Verdadero. Se representa por $\mathcal{I}_A \models F$. Dado un conjunto de fórmulas S, una interpretación que sea realización de todas las fórmulas de S se dice que es una realización de S.

$\mathcal{I}$ es un modelo de F si, para toda asignación A en $\mathcal{I}$, $\mathcal{I}_A(F) =$ Verdadero. Se representa por $\mathcal{I} \models F$. Si S es un conjunto de fórmulas y $\mathcal{I} \models F$ para toda fórmula F de S se dice que $\mathcal{I}$ es un modelo de S.

Sea F una fórmula del lenguaje:

- F es válida si para toda estructura $\mathcal{I}$ del lenguaje y toda asignación A en $\mathcal{I}$ se tiene que $\mathcal{I}_A(F) =$ Verdadero. Se representa por $\models F$.

- F es satisfacible si tiene alguna realización. Si un conjunto de fórmulas S tiene alguna realización se dice que es consistente.
- F es insatisfacible si no tiene ninguna realización. Si un conjunto de fórmulas S no tiene ninguna realización se dice que es inconsistente.

Se puede probar la siguiente propiedad: una fórmula F es válida si y sólo si $\neg F$ es insatisfacible.

Al igual que en lógica proposicional, si $F \Leftrightarrow G$ es lógicamente válida decimos que F y G son lógicamente equivalentes. Se dice que F es consecuencia lógica de S si todos los modelos de S lo son de F, y lo notamos $S \models F$. Usaremos la notación $S \models_M F$ para indicar que M es un modelo específico de F siempre que lo sea de S.

Sea $S = \{F_1, ..., F_n\}$ finito, donde las F_i son fórmulas cerradas. Entonces $\{F_1, ..., F_n\} \models F$ es equivalente a $F_1 \wedge ... \wedge F_n \Rightarrow F$.

No es posible implementar un algoritmo para decidir la validez o la satisfacibilidad de la lógica de primer orden directamente a través de la semántica. No tenemos forma de verificar si una fórmula de primer orden es satisfacible en una interpretación con un dominio infinito. Sin embargo, se dará posteriormente un algoritmo para transformar fórmulas manteniendo la equisatisfabilidad, y abordaremos el problema entonces.

4.3. Operaciones sintácticas sobre fórmulas

Introduciremos algunas operaciones que usaremos posteriormente para la transformación de fórmulas en formas normales.

Sea F una fórmula de primer orden y $x_1, ..., x_n$ sus variables libres. Se denomina cierre universal de F a $\forall x_1, ..., x_n.F$. Se demuestra que una fórmula es válida si y sólo si su cierre universal lo es, y es más conveniente trabajar con fórmulas cerradas. Así, como explicamos antes, si todas las fórmulas que intervienen son cerradas tenemos:

$$\begin{aligned} \{F_1, ..., F_n\} \models F \quad &\Leftrightarrow \quad F_1 \wedge ... \wedge F_n \Rightarrow F \\ F \text{ es válida} \quad &\Leftrightarrow \quad \neg F \text{ es insatisfacible} \end{aligned}$$

Implementamos en OCaml el concepto de cierre universal:

```
let generalize fm = itlist mk_forall (fv fm) fm;;
```

Se define también el concepto de cierre existencial de una fórmula F como $\exists x_1 ... x_n.F$, donde x_1,...,x_n son las variables libres de F. Se demuestra que una fórmula es satisfacible si y sólo si su cierre existencial lo es.

Otra operación que necesitamos definir es la sustitución de una variable por un término en otro término o una fórmula. Por ejemplo, sustituir x por 1 en $x < 2 \Rightarrow x \leq y$ para obtener $1 < 2 \Rightarrow 1 \leq y$. Una sustitución σ de Σ es una aplicación $\sigma : Var \longrightarrow Term(\Sigma)$. Denotaremos las sustituciones por $[x_1/t_1, ..., x_n/t_n]$.

Especificaremos la sustitución como una función `sfn` de los nombres de las variables en términos. Para las variables que no queremos cambiar éstos pueden ser indefinidos o simplemente la misma variable.

```
let rec tsubst sfn tm =
  match tm with
    Var x -> tryapplyd sfn x tm
  | Fn(f,args) -> Fn(f,map (tsubst sfn) args);;
```

Para aplicar la sustitución a fórmulas debemos tener más cuidado, debido a las variables ligadas. Sin embargo, aún evitando las sustituciones de dichas variables, corremos el riesgo de sustituir una variable libre por otra que en nuestra fórmula esté ligada. Por ejemplo, reemplazar y por x en la fórmula $\exists x.x + 1 = y$ da como resultado $\exists x.x + 1 = x$, que no es lo que queríamos, pues hemos obtenido una fórmula cerrada. Para evitar esto, en primer lugar, vamos a renombrar las variables ligadas que sean necesarias. Implementamos una función en Ocaml que añada caracteres a la variable hasta que sea distinta de todas las variables de un lista dada.

```
let rec variant x vars =
  if mem x vars then variant (x^"'") vars else x;;
```

Por ejemplo:

```
# variant "x" ["y"; "z"];;
- : string = "x"
# variant "x" ["x"; "y"];;
- : string = "x'"
# variant "x" ["x"; "x'"];;
- : string = "x''"
```

Ahora, la definición de sustitución empieza con una serie de sencillas recursiones estructurales. Sin embargo, los dos casos más delicados de fórmulas cuantificadas, $\forall x.F$ y $\exists x.F$ son son tratados por otra función mutuamente recursiva `substq`.

```
let rec subst subfn fm =
  match fm with
    False -> False
  | True -> True
  | Atom(R(p,args)) -> Atom(R(p,map (tsubst subfn) args))
  | Not(p) -> Not(subst subfn p)
  | And(p,q) -> And(subst subfn p,subst subfn q)
  | Or(p,q) -> Or(subst subfn p,subst subfn q)
  | Imp(p,q) -> Imp(subst subfn p,subst subfn q)
  | Iff(p,q) -> Iff(subst subfn p,subst subfn q)
  | Forall(x,p) -> substq subfn mk_forall x p
  | Exists(x,p) -> substq subfn mk_exists x p

and substq subfn quant x p =
  let x' = if exists (fun y -> mem x (fvt(tryapplyd subfn y (Var y))))
                     (subtract (fv p) [x])
           then variant x (fv(subst (undefine x subfn) p)) else x in
  quant x' (subst ((x |-> Var x') subfn) p);;
```

Esta función `substq` comprueba si existe alguna variable libre $y \neq x$ tal que aplicar la sustitución a y devuelve un término con x libre. Si es así, toma una nueva variable x' que no entrará en conflicto con ninguna de las otras sustituciones en F. Entonces se aplica la sustitución a p cambiando x por x'.
Por ejemplo:

```
# subst ("y" |=> Var "x") <<forall x. x = y>>;;
- : fol formula = <<forall x'. x' = x>>
# subst ("y" |=> Var "x") <<forall x x'. x = y ==> x = x'>>;;
- : fol formula = <<forall x' x''. x' = x ==> x' = x''>>
```

Obtenemos una propiedad importante: Si una fórmula es válida cualquier sustitución de dicha fórmula lo es.

Una fórmula F se dice que está en forma rectificada si ninguna variable aparece libre y ligada simultáneamente y cada cuantificador se refiere a una variable diferente.

4.4. Forma normal prenexa

Una fórmula de primer orden F se dice que está en forma normal prenexa o PNF (del inglés, prenex normal form) si es de la forma $(Q_1x_1)\ldots(Q_nx_n)\,G$, donde $Q_i \in \{\forall, \exists\}$, $n \geq 0$ y G no tiene cuantificadores. $(Q_1x_1)\ldots(Q_nx_n)$ se llama el prefijo de F y G se llama la matriz de F. Mostraremos en esta sección como transformar una fórmula en otra lógicamente equivalente en forma normal prenexa.

En primer lugar, debemos eliminar los cuantificadores vacuos en una fórmula. Esto es, aquellos que cuantifican una variable ya cuantificada o una variable que

no aparece en la fórmula. Para ello aplicaremos el siguiente resultado: si x no es una variable libre de F, tanto $\forall x.F$ como $\exists x.F$ son lógicamente equivalentes a F. Para definir la primera simplificación de fórmulas de primer orden, usando este resultado, haremos uso además de la función definida para lógica proposicional `psimplify1`:

```
let simplify1 fm =
  match fm with
    Forall(x,p) -> if mem x (fv p) then fm else p
  | Exists(x,p) -> if mem x (fv p) then fm else p
  | _ -> psimplify1 fm;;
```

Y podemos aplicarlo de manera recursiva a cada subfórmula:

```
let rec simplify fm =
  match fm with
    Not p -> simplify1 (Not(simplify p))
  | And(p,q) -> simplify1 (And(simplify p,simplify q))
  | Or(p,q) -> simplify1 (Or(simplify p,simplify q))
  | Imp(p,q) -> simplify1 (Imp(simplify p,simplify q))
  | Iff(p,q) -> simplify1 (Iff(simplify p,simplify q))
  | Forall(x,p) -> simplify1(Forall(x,simplify p))
  | Exists(x,p) -> simplify1(Exists(x,simplify p))
  | _ -> fm;;
```

Simplifiquemos como ejemplo la fórmula $(\forall x\ y.\ P(x) \vee (P(y) \wedge Falso)) \Rightarrow \exists z.Q$:

```
# simplify <<(forall x y. P(x) \/ (P(y) /\ false)) ==> exists z. Q>>;;
- : fol formula = <<(forall x. P(x)) ==> Q>>
```

A continuación eliminamos los bicondicionales y las implicaciones e interiorizamos las negaciones. Para ello aplicamos las leyes de De Morgan que ya enunciamos cuando transformábamos las fórmulas proposicionales en formas normales negativas, y las leyes de De Morgan para cuantificadores, que enunciamos a continuación:

$$\begin{array}{rcl} \neg\,(\forall x.F) & \Leftrightarrow & \exists x.\neg F \\ \neg\,(\exists x.F) & \Leftrightarrow & \forall x.\neg F \end{array}$$

Lo implementamos en OCaml:

```
let rec nnf fm =
  match fm with
    And(p,q) -> And(nnf p,nnf q)
  | Or(p,q) -> Or(nnf p,nnf q)
  | Imp(p,q) -> Or(nnf(Not p),nnf q)
  | Iff(p,q) -> Or(And(nnf p,nnf q),And(nnf(Not p),nnf(Not q)))
  | Not(Not p) -> nnf p
  | Not(And(p,q)) -> Or(nnf(Not p),nnf(Not q))
  | Not(Or(p,q)) -> And(nnf(Not p),nnf(Not q))
  | Not(Imp(p,q)) -> And(nnf p,nnf(Not q))
  | Not(Iff(p,q)) -> Or(And(nnf p,nnf(Not q)),And(nnf(Not p),nnf q))
  | Forall(x,p) -> Forall(x,nnf p)
  | Exists(x,p) -> Exists(x,nnf p)
  | Not(Forall(x,p)) -> Exists(x,nnf(Not p))
  | Not(Exists(x,p)) -> Forall(x,nnf(Not p))
  | _ -> fm;;
```

Por ejemplo:

```
# nnf <<(forall x. P(x))
      ==> ((exists y. Q(y)) <=> exists z. P(z) /\ Q(z))>>;;
  - : fol formula =
<<(exists x. ~P(x)) \/
  (exists y. Q(y)) /\ (exists z. P(z) /\ Q(z)) \/
  (forall y. ~Q(y)) /\ (forall z. ~P(z) \/ ~Q(z))>>
```

Tras estas transformaciones pasamos a la parte realmente distintiva de la forma normal prenexa: exteriorizar los cuantificadores. Para ello hacemos uso de las siguientes equivalencias, suponiendo que x no es libre en G:

- $(\forall x.F) \wedge G \equiv \forall x.\,(F \wedge G)$
- $(\forall x.F) \vee G \equiv \forall x.\,(F \vee G)$
- $(\exists x.F) \wedge G \equiv \exists x.\,(F \wedge G)$
- $(\exists x.F) \vee G \equiv \exists x.\,(F \vee G)$
- $G \wedge (\forall x.F) \equiv \forall x.\,(G \wedge F)$
- $G \vee (\forall x.F) \equiv \forall x.\,(G \vee F)$
- $G \wedge (\exists x.F) \equiv \exists x.\,(G \vee F)$
- $G \vee (\exists x.F) \equiv \exists x.\,(G \vee F)$

Debemos tener cuidado al exteriorizar los cuantificadores con las variables libres. Por ejemplo:

$$P(x) \wedge (\exists x.Q(x)) \text{ no es lógicamente equivalente a } \exists x.\ (P(x), \wedge Q(x))$$

pues la variable x es libre en la primera fórmula pero no en la segunda. En estos casos, renombraremos la variable ligada, mediante la sustitución de ésta por una variable que no aparezca en la fórmula. Esto es, rectificaremos la fórmula. En el ejemplo anterior la primera fórmula sí sería lógicamente equivalente a $\exists y.\,(P\,(x) \wedge Q\,(y))$.

Para exteriorizar los cuantificadores que ocurren como las subfórmulas inmediatas de una conjunción o una disyunción, definimos la siguiente función en Ocaml:

```
let rec pullquants fm =
  match fm with
    And(Forall(x,p),Forall(y,q)) ->
                          pullq(true,true) fm mk_forall mk_and x y p q
  | Or(Exists(x,p),Exists(y,q)) ->
                          pullq(true,true) fm mk_exists mk_or x y p q
  | And(Forall(x,p),q) -> pullq(true,false) fm mk_forall mk_and x x p q
  | And(p,Forall(y,q)) -> pullq(false,true) fm mk_forall mk_and y y p q
  | Or(Forall(x,p),q) ->  pullq(true,false) fm mk_forall mk_or x x p q
  | Or(p,Forall(y,q)) ->  pullq(false,true) fm mk_forall mk_or y y p q
  | And(Exists(x,p),q) -> pullq(true,false) fm mk_exists mk_and x x p q
  | And(p,Exists(y,q)) -> pullq(false,true) fm mk_exists mk_and y y p q
  | Or(Exists(x,p),q) ->  pullq(true,false) fm mk_exists mk_or x x p q
  | Or(p,Exists(y,q)) ->  pullq(false,true) fm mk_exists mk_or y y p q
  | _ -> fm
```

la cual llama a otra función mutuamente recursiva a ésta, para englobar varios casos similares:

```
and pullq(l,r) fm quant op x y p q =
  let z = variant x (fv fm) in
  let p' = if l then subst (x |=> Var z) p else p
  and q' = if r then subst (y |=> Var z) q else q in
  quant z (pullquants(op p' q'));;
```

Así sustituiríamos las variables necesarias para exteriorizar los cuantificadores. Mediante `l` y `r` indicamos si hacemos la sustitución en la subfórmula izquierda o derecha de la conectiva lógica, respectivamente. Mediante la función `variant` se obtiene una variable que no haya aparecido aún en la fórmula, y hacemos la sustitución con dicha variable. Después, vuelve a llamar a la función `pullquants`.

Aplicamos esto a la fórmula completa de manera recursiva.

```
let rec prenex fm =
  match fm with
    Forall(x,p) -> Forall(x,prenex p)
  | Exists(x,p) -> Exists(x,prenex p)
  | And(p,q) -> pullquants(And(prenex p,prenex q))
  | Or(p,q) -> pullquants(Or(prenex p,prenex q))
  | _ -> fm;;
```

Por último, simplificando la fórmula y haciendo uso de la forma normal negativa definida para fórmulas de primer orden, implementamos la forma normal prenexa.

```
let pnf fm = prenex(nnf(simplify fm));;
```

Como ejemplo calculemos la forma normal prenexa de la fórmula

$$(\forall x.(P(x) \vee R(y))) \Rightarrow \exists y\ z.(Q(y) \vee \neg(\exists z.(P(z) \wedge Q(z))))$$

En primer lugar simplifiquemos la fórmula y calculemos su forma normal negativa:

```
# nnf(simplify <<(forall x. P(x) \/ R(y)) ==> exists y z.
Q(y) \/ ~(exists z. P(z) /\ Q(z)) >>);;
- : fol formula =
<<(exists x. ~P(x) /\ ~R(y)) \/
  (exists y. Q(y) \/ (forall z. ~P(z) \/ ~Q(z)))>>
```

Podemos observar que en la fórmula resultante, la variable y tiene tanto apariciones libres como ligadas. Por tanto debemos rectificarla.

Para ello basta sustituir la variable ligada y por x, ya que no entran en conflicto, y agrupar los dos cuantificadores existenciales en uno sólo. Así, obtenemos:

```
# pnf <<(forall x. P(x) \/ R(y)) ==> exists y z.
Q(y) \/ ~(exists z. P(z) /\ Q(z)) >>;;
  - : fol formula =
<<exists x. forall z. ~P(x) /\ ~R(y) \/ Q(x) \/ ~P(z) \/ ~Q(z)>>
```

4.5. Forma de Skolem

Una fórmula F se dice que está en forma normal de Skolem si es de la forma $\forall x_1 ... \forall x_n.G$, donde $n \geq 0$ y G no tiene cuantificadores.

Enunciamos a continuación dos propiedades necesarias en el algoritmo para el cálculo de la forma normal de Skolem:

- Si a es una constante que no ocurre en F, entonces $\exists x.F \approx F\,[x/a]$. a se denomina constante de Skolem.

- Si g es un símbolo de función n-aria que no ocurre en F, entonces:

$$\forall x_1 ... \forall x_n \exists x.F \approx \forall x_1 ... \forall x_n F\,[x/g(x_1, ..., x_n)]$$

  g se denomina función de Skolem.

Aquí $F \approx G$ simboliza F y G son equisatisfacibles. Recordamos la definición de equisatisfacibilidad: F y G son equisatisfacibles si F es satisfacible si y sólo si G es satisfacible.

Implementamos en Ocaml a continuación un procedimiento para obtener las funciones que aparecen en una fórmula de primer orden. Las identificaremos como pares nombre-aridad.

```
let rec funcs tm =
  match tm with
    Var x -> []
  | Fn(f,args) -> itlist (union ** funcs) args [f,length args];;

let functions fm =
  atom_union (fun (R(p,a)) -> itlist (union ** funcs) a []) fm;;
```

En el algoritmo que vamos a implementar para obtener la forma normal de Skolem de una fórmula de primer orden F no transformaremos previamente F en forma normal prenexa. Esto se debe a que el algoritmo para el cálculo de la forma normal prenexa exterioriza los cuantificadores existenciales, introduciendo las variables libres de la fórmula original en el alcance de dichos cuantificadores, por lo que las funciones de Skolem necesitarían más argumentos.

Por ejemplo, la forma normal de Skolem de la fórmula

$$\forall xz.(x = z \vee \exists y.(x \cdot y = 1))$$

es

$$\forall xz.(x = z \vee x \cdot f(x) = 1)$$

mientras que si la transformamos primero en forma normal prenexa:

$$\forall xz.\ \exists y.(x = z \vee x \cdot y = 1)$$

su forma normal de Skolem es

$$\forall xz.(x = z \vee x \cdot f(x, z) = 1)$$

Sin embargo, para utilizar las funciones de Skolem necesitamos introducir las negaciones en la fórmula original. Veamos un ejemplo:

$$(\exists y.P(y)) \wedge \neg(\exists x.P(x))$$

es insatisfacible mientras que si transformamos la subfórmula derecha en forma normal de Skolem sin introducir previamente la negación, la fórmula resultante obtenida es satisfacible:

$$(\exists y.P(y)) \wedge \neg P(c) \tag{4.1}$$

Es por ello que, para mantener la satisfacibilidad o instisfacibilidad de la fórmula original, en primer lugar transformaremos las fórmulas en forma normal negativa.

Para transformar una fórmula en forma normal negativa descenderemos a través de ella eliminando los cuantificadores existenciales mediante las funciones de Skolem y transformando las subfórmulas en forma normal de Skolem. Evitaremos usar funciones de Skolem con el mismo nombre que las funciones que ya aparezcan en nuestra fórmula, aunque tengan distinta aridad.

```
let rec skolem fm fns =
  match fm with
    Exists(y,p) ->
        let xs = fv(fm) in
        let f = variant (if xs = [] then "c_"^y else "f_"^y) fns in
        let fx = Fn(f,map (fun x -> Var x) xs) in
        skolem (subst (y |=> fx) p) (f::fns)
  | Forall(x,p) -> let p',fns' = skolem p fns in Forall(x,p'),fns'
  | And(p,q) -> skolem2 (fun (p,q) -> And(p,q)) (p,q) fns
  | Or(p,q) -> skolem2 (fun (p,q) -> Or(p,q)) (p,q) fns
  | _ -> fm,fns

and skolem2 cons (p,q) fns =
  let p',fns' = skolem p fns in
  let q',fns'' = skolem q fns' in
  cons(p',q'),fns'';;
```

La función `skolem` devuelve un par formado por la función original en forma normal de Skolem y por las funciones que aparecen en el primer elemento del par. La función `skolem2` es una función auxiliar para agrupar los casos de las conectivas binarias `And` y `Or`.

La función principal del algoritmo, que implementamos a continuación, simplifica la fórmula y la transforma en forma normal negativa. Después aplica la función `skolem` con un conjunto inicial apropiado de símbolos de funciones a evitar.

```
let askolemize fm =
  fst(skolem (nnf(simplify fm)) (map fst (functions fm)));;
```

Implementamos las funciones siguientes para obtener la forma normal de Skolem de una fórmula sin los cuantificadores universales, previamente exteriorizados, ni los existenciales, eliminados mediante la función `skolem`:

```
let rec specialize fm =
  match fm with
    Forall(x,p) -> specialize p
  | _ -> fm;;

let skolemize fm = specialize(pnf(askolemize fm));;
```

Veamos como ejemplo qué obtenemos al aplicar las funciones anteriores a la fórmula $\exists y.((x < y) \Rightarrow \forall u\ \exists v.\ x * u < y * v)$:

```
# askolemize <<exists y. x < y ==> forall u. exists v. x * u < y * v>>;;
- : fol formula =
<<~x < f_y(x) \/ (forall u. x * u < f_y(x) * f_v(u,x))>>

# skolemize <<exists y. x < y ==> forall u. exists v. x * u < y * v>>;;
- : fol formula = <<~x < f_y(x) \/ x * u < f_y(x) * f_v(u,x)>>

# askolemize ( pnf <<exists y. x < y ==> forall u.
  exists v. x * u < y * v>>);;
- : fol formula =
<<forall u. ~x < f_y(x) \/ x * u < f_y(x) * f_v(u,x)>>
```

En el primer caso, obtenemos la forma normal de Skolem de una fórmula sin transformarla previamente en forma normal prenexa y sin eliminar los cuantificadores al final. En el segundo caso hemos eliminado los cuantificadores de la fórmula. En el último caso hemos transformado la fórmula previamente en forma normal prenexa y después hemos aplicado el algoritmo para el cálculo de la forma normal de Skolem.

4.6. Teorema de Herbrand

Sea L el lenguaje de primer orden sin igualdad. Definimos el universo de Herbrand de L como el conjunto de los términos básicos de L, esto es, todos los términos que se pueden construir a partir de las constantes y los símbolos de función del lenguaje sin usar variables. Se representa por $UH(L)$. Si el lenguaje no tiene constantes añadimos una constante a para hacer el universo de Hebrand no vacío.

Sea C el conjunto de constantes de L y F_n el conjunto de símbolos de función n-aria de L.

$$UH(L) = \cup_{i \geq 0} H_i(L)$$

donde $H_i(L)$ es el nivel i del $UH(L)$ definido por

$$\begin{aligned} H_0(L) &= \begin{cases} C, & \text{si } C \neq \emptyset \\ \{a\}, & \text{en caso contrario} \end{cases} \\ H_{i+1}(L) &= H_i(L) \cup \{f(t_1, ..., t_n) : f \in F_n \text{ y } t_1, ..., t_n \in H_i(L)\} \end{aligned}$$

Por definición, $UH(L)$ es finito si y sólo si L no tiene símbolos de función.

La base de Herbrand de L es el conjunto de los átomos básicos de L. Se representa por $BH(L)$. Si R_n es el conjunto de símbolos de relación n-aria de L, entonces

$$BH(L) = \{P(t_1, ..., t_n) : P \in R_n \text{ y } t_1, ..., t_n \in UH(L)\}$$

Por definición, $BH(L)$ es finito si y sólo si L no tiene símbolos de función.

4.6.1. Cláusulas de primer orden

Un átomo es una fórmula atómica, es decir, un símbolo de relación n-ario aplicado a n términos. Un literal es un átomo o su negación.

Como ya definimos, una cláusula es un conjunto de literales, que denotamos C. En lógica de primer orden, la fórmula correspondiente a la cláusula

$$C = \{L_1, ..., L_m\} \text{ es } \forall x_1 ... \forall x_p.(L_1 \vee ... \vee L_m)$$

donde $x_1, ..., x_p$ son las variables libres de $L_1 \vee ... \vee L_m$. La fórmula correspondiente al conjunto de cláusulas $\{\{L_{11}, ..., L_{1m_1}\}, ..., \{L_{n1}, ..., L_{nm_n}\}\}$ cuyas variables libres son $x_1, ..., x_p$, es

$$\forall x_1 ... \forall x_p((L_{11} \vee ... \vee L_{1m_1}) \wedge ... \wedge (L_{n1} \vee ... \vee L_{nm_n}))$$

Una forma clausal de una fórmula F es un conjunto de cláusulas S tal que $F \approx S$. Para obtener la forma clausal de una fórmula de primer orden hacemos uso de la forma normal de Skolem y de los algoritmos para transformar una fórmula en forma normal conjuntiva.

Por ejemplo,

$$F = \neg\exists x.(P(x) \Rightarrow \forall x.P(x)) \approx$$
$$\approx \forall x.(P(x) \wedge \neg P(f(x))) \equiv$$
$$\equiv \{\{P(x)\}, \{\neg P(f(x))\}\}$$

Esta es una forma clausal de la fórmula F.

4.6.2. Extensiones de Herbrand

Antes de enunciar el Teorema de Herbrand debemos introducir varios nuevos conceptos: las instancias básicas de una cláusula y las extensiones de Herbrand.

Recordemos que una sustitución σ de L es una aplicación $\sigma : Var \longrightarrow Term(L)$. Sea $C = \{L_1, ..., L_n\}$ una cláusula de L y σ una sustitución de L. Entonces

$$C\sigma = \{L_1\sigma, ..., L_n\sigma\}$$

es una instancia de C.

Por ejemplo, si $C = \{P(x, a), \neg P(x, f(y))\}$, una instancia de C es

$$C\,[x/f(a), y/f(a)] = \{P(f(a), a), \neg P(f(a), f(f(a)))\}$$

$C\sigma$ es una instancia básica de C si todos los literales de $C\sigma$ son básicos, es decir, no intervienen variables. En nuestro ejemplo anterior, $C\sigma$ es una instancia básica de C.

La extensión de Herbrand de un conjunto de cláusulas S es el conjunto de fórmulas

$$EH(S) = \{C\sigma : C \in S \text{ y, para toda variable } x \text{ en } C,\ \sigma(x) \in UH(S)\}.$$

Se verifica $EH(L) = \cup_{i \geq 0} EH_i(L)$, donde $EH_i(L)$ es el nivel i de la $EH(L)$:

$$EH_i(S) = \{C\sigma : C \in S \text{ y, para toda variable } x \text{ en } C,\ \sigma(x) \in UH_i(S)\}$$

Teorema de Herbrand: Sea S un conjunto de cláusulas. S es consistente si y sólo si $EH(S)$ es consistente (en el sentido de la lógica proposicional).

Se puede probar el siguiente resultado: Si S es un conjunto de cláusulas, son equivalentes:

1. S es inconsistente.
2. $EH(S)$ tiene un subconjunto finito inconsistente (en el sentido de la lógica proposicional).
3. Para algún i, $EH_i(S)$ es inconsistente (en el sentido de la lógica proposicional).

Esto nos da un método para verificar si una fórmula F es insatisfacible, comprobando si cada $EH_i(S)$ es insatisfacible en el sentido proposicional, donde $S \approx F$. Si F es insatisfacible, entonces este procedimiento es finito. Sin embargo, si la fórmula F es satisfacible, este procedimiento es infinito, y por ello se denomina procedimiento de semidecisión.

Sin embargo, si nos restringimos a conjuntos más pequeños de fórmulas, podemos ser capaces de encontrar un procedimiento de decisión para éstas. Por ejemplo, la extensión de Herbrand de las fórmulas en las que no aparecen símbolos de función es finita y, por tanto, el procedimiento para decidir la satisfacibilidad de estas fórmulas es finito.

No implementaremos esto en Ocaml, pues el objetivo de esta sección era mostrar que no existen procedimientos para decidir la satisfacibilidad de las fórmulas de primer orden, dotando este hecho de importancia al algoritmo de eliminación de cuantificadores.

Capítulo 5

Eliminación de cuantificadores

Como ya hemos comentado en el capítulo anterior, el problema de la validez en lógica de primer orden es indecidible. Nos preguntamos entonces si existe algún algoritmo para decidir la validez o satisfacibilidad de conjuntos más pequeños de fórmulas. En este contexto surge la eliminación de cuantificadores.

Antes de hablar de dicho algoritmo introduciremos algunas nociones básicas sobre estos conjuntos más pequeños de fórmulas a los que vamos a restringirnos: las teorías de primer orden.

5.1. Teorías de primer orden

Una teoría de primer orden T, o simplemente una teoría, se define a partir de un lenguaje de primer orden Σ, es decir, un conjunto de símbolos de constantes, funciones y predicados; y un conjunto de fórmulas cerradas en las que sólo aparecen los símbolos de Σ. Estas fórmulas son llamadas los axiomas no lógicos de T.

Una Σ-fórmula es una fórmula construida a partir de símbolos de constantes, funciones y predicados de Σ, así como de variables, conectivas lógicas y cuantificadores.

Una interpretación I que satisface todos los axiomas de T se denomina T-interpretación.

Una Σ-fórmula F es válida en la teoría T, o T-válida, si toda T-interpretación satisface F. Lo denotamos $T \models F$.

Una Σ-fórmula F es satisfacible en la teoría T, o T-satisfacible, si hay alguna T-interpretación que satisface F.

Describimos a continuación algunas propiedades importantes de las teorías:

- Una teoría T es consistente si hay al menos una T-interpretación. En particular, en una teoría consistente T, no existe ninguna Σ-fórmula F tal que se verifique a la vez tanto $T \models F$ como $T \models \neg F$.

- Una teoría T es completa si para cada Σ-fórmula cerrada F, $T \models F$ o $T \models \neg F$.
- Una teoría T es decidible si existe un algoritmo que dada cualquier Σ-fórmula F decide si $T \models F$.

Un fragmento de una teoría T es un subconjunto sintácticamente restringido de fórmulas de la teoría. Un fragmento de T es decidible si $T \models F$ es decidible para cada Σ-fórmula F que obedece las restricciones sintácticas del fragmento.

5.2. Procedimiento de eliminación de cuantificadores

La eliminación de cuantificadores es la construcción de una fórmula sin cuantificadores equivalente a una dada. Cabe notar que no todas las teorías admiten un procedimiento de eliminación de cuantificadores. Formalmente una teoría admite eliminación de cuantificadores si existe un algoritmo que dada cualquier fórmula F en esta teoría devuelve otra fórmula equivalente a ésta F' sin cuantificadores, tal que el conjunto de las variables libres de F' está contenido en el de las variables libres de F.

Una teoría T que admita eliminación de cuantificadores será decidible si el fragmento 'libre de cuantificadores' de dicha teoría lo es.

En primer lugar, veamos que basta probar que una teoría admite eliminación de cuantificadores sólo para las fórmulas de la forma $F = \exists x.(\alpha_0 \wedge ... \wedge \alpha_n)$, donde cada α_i es un literal que contiene a x. Lo probaremos por inducción en las fórmulas.

- Caso base: claramente una fórmula atómica es equivalente a una fórmula sin cuantificadores: ella misma.
- Hipótesis de inducción: supongamos que G y H son equivalentes a fórmulas sin cuantificadores G' y H', respectivamente.
- Probémoslo en general.
  - Es trivial para las conectivas lógicas:
    - $T \Vdash \neg G \Leftrightarrow \neg G'$,
    - $T \Vdash (G \wedge H) \Leftrightarrow (G' \wedge H')$,
    - $T \Vdash (G \vee H) \Leftrightarrow (G' \vee H')$,
    - $T \Vdash (G \Rightarrow H) \Leftrightarrow (G' \Rightarrow H')$,
    - $T \Vdash (G \Leftrightarrow H) \Leftrightarrow (G' \Leftrightarrow H')$.
  - Como $\forall x.G$ es equivalente a $\neg(\exists x.\neg G)$, basta probarlo para las fórmulas de la forma $\exists x.G$.

- Por la hipótesis de inducción, $\exists x.G$ es equivalente a $\exists x.G'$, donde G' es una fórmula sin cuantificadores. Ahora, transformando G' en su forma normal disyuntiva tendríamos

$$T \Vdash (\exists x.G') \Leftrightarrow (\exists x.(\gamma_0 \vee ... \vee \gamma_m))$$

donde cada γ_i es una conjunción de literales. Entonces

$$T \Vdash (\exists x.G') \Leftrightarrow (\exists x.\gamma_0) \vee ... \vee (\exists x.\gamma_m)$$

Como para cada fórmula $\exists x.\gamma_i$ podemos encontrar una fórmula sin cuantificadores equivalente a ella, habríamos encontrado una fórmula sin cuantificadores equivalente a $\exists x.G$.

Usaremos la demostración anterior para implementar en Ocaml un procedimiento de eliminación de cuantificadores para cualquier tipo de fórmula, dado uno para fórmulas de la forma $F = \exists x.(\alpha_0 \wedge ... \wedge \alpha_n)$.

Definamos una función que elimine el cuantificador existencial de la fórmula

$$\exists x.(\alpha_0 \wedge ... \wedge \alpha_n)$$

Sea `bfn` un procedimiento para fórmulas de este tipo, específico de cada teoría que admite eliminación de cuantificadores; `p` una conjunción de literales en la que algunos no contienen a x, y consideremos la fórmula $\exists x.p$. La siguiente función hace una partición del conjunto de todos los literales en aquellos que contienen a x (`ycjs`) y aquellos que no lo contienen (`ncjs`), y separa este último antes de aplicar `bfn` al resto, usando implícitamente la equivalencia

$$(\exists x.p \wedge q(x)) \Leftrightarrow p \wedge \exists x.q(x)$$

```
let qelim bfn x p =
  let cjs = conjuncts p in
  let ycjs,ncjs = partition (mem x ** fv) cjs in
  if ycjs = [] then p else
  let q = bfn (Exists(x,list_conj ycjs)) in
  itlist mk_and ncjs q;;
```

5.2.1. Reducción del alcance de los cuantificadores

Para hacer más fácil la tarea de eliminar los cuantificadores de una fórmula, intentaremos en primer lugar reducir su alcance. Esto es, usar equivalencias para aplicar los cuantificadores sólo a las subfórmulas en las que aparezca la variable cuantificada y ésta sea libre. De esta manera simplificaremos el procedimiento de eliminación de cuantificadores.

Definamos una función `separate` que transforme una fórmula $\exists x.(F_1 \wedge ... \wedge F_n)$ en

$$(\exists x.(F_i \wedge ... \wedge F_j)) \wedge (F_k \wedge ... \wedge F_l),$$

donde los F_i,..., F_j son las fórmulas con x libre y F_k,..., F_l son las demás. Las conjunciones de la fórmula de entrada son presentadas como un conjunto `cjs`:

```
let separate x cjs =
  let yes,no = partition (mem x ** fv) cjs in
  if yes = [] then list_conj no
  else if no = [] then Exists(x,list_conj yes)
  else And(Exists(x,list_conj yes),list_conj no);;
```

Definimos ahora una función `pushquant`, que dada una variable x y una fórmula F transforma la fórmula $\exists x.F$ en una equivalente con el alcance del cuantificador existencial reducido. En primer lugar, si x no es libre en la fórmula F, la respuesta es F. En otro caso, transformamos F en forma normal disyuntiva, siendo de la forma $\exists x.(C_1 \vee ... \vee C_n)$, donde cada C_i es una conjunción de literales. Lo transformamos ahora en $(\exists x.C_1) \vee ... \vee (\exists x.C_n)$, y aplicamos a cada miembro de la disyunción la función `separate`:

```
let rec pushquant x p =
  if not (mem x (fv p)) then p else
  let djs = purednf(nnf p) in
  list_disj (map (separate x) djs);;
```

Implementamos a continuación una función que transforma la fórmula $\forall x.G$ en

$$\neg(\exists x.\neg G)$$

y aplica `pushquant` posteriormente. Suponemos que la fórmula inicial está en forma normal negativa, reduciendo así los casos a tratar:

```
let rec miniscope fm =
  match fm with
    Not p -> Not(miniscope p)
  | And(p,q) -> And(miniscope p,miniscope q)
  | Or(p,q) -> Or(miniscope p,miniscope q)
  | Forall(x,p) -> Not(pushquant x (Not(miniscope p)))
  | Exists(x,p) -> pushquant x (miniscope p)
  | _ -> fm;;
```

5.2.2. Función principal

Ahora definimos la función principal. Notemos que es de orden superior, esto es, toma como argumentos varias funciones y devuelve otra función. Sus argumentos son:

- `afn`: es una función propia de cada teoría, que se aplica a una lista de variables y a una fórmula. Supongamos por el momento que `afn vars fm` simplemente devuelve su segundo argumento `fm` sin cambios.
- `nfn`: es una función que transforma una fórmula en forma normal disyuntiva.
- `qfn`: es la función `bfn` que aparece en la anterior `qelim`. Es específica de cada teoría y elimina los cuantificadores de las fórmulas de la forma

  $$\exists x.(\alpha_0(x) \wedge ... \wedge \alpha_n(x))$$

  donde los α_i son literales.

Cabe destacar que aunque estemos definiendo un procedimiento general de eliminación de cuantificadores, como las funciones anteriores son propias de cada teoría, ésta admitirá eliminación de cuantificadores si podemos definir dichas funciones para la teoría.

Veamos cómo actúa la siguiente función `lift_qelim` sobre una fórmula `fm`:

1. Simplificamos la fórmula con la función `miniscope` para aplicar el algoritmo a la subfórmula más pequeña posible.
2. Distingamos varios casos según el tipo de fórmula:
   a) Para las conectivas lógicas se lleva a cabo recursivamente el mismo procedimiento: la función auxiliar `qelift` es aplicada a cada subfórmula.
   b) Si la fórmula está universalmente cuantificada, transformamos el cuantificador universal en existencial usando las Leyes de De Morgan.
   c) El caso más interesante entonces es el de una fórmula existencialmente cuantificada: $\exists x.p$.
      1) Se aplica de manera recursiva el procedimiento de eliminación de cuantificadores a p, con lo que obtenemos una fórmula sin cuantificadores equivalente a p. Denominémosla p'.
      2) Obtenemos la forma normal disyuntiva de p' haciendo uso de `nnf`. Definimos `djs` como el conjunto de todas estas conjunciones.
      3) Aplicamos la función `qelim` a cada elemento de `djs`, tomando el argumento `qfn` como `bfn`, usando implícitamente la equivalencia:

         $$(\exists x.(D_1(x) \vee ... \vee D_n(x)) \Leftrightarrow (\exists x.D_1(x)) \vee ... \vee (\exists x.D_n(x)).$$

Devuelve la disyunción de los elementos del conjunto anterior.

3. Por último simplificamos la fórmula resultante.

```
let lift_qelim afn nfn qfn =
  let rec qelift vars fm =
    match fm with
    | Atom(R(_,_)) -> afn vars fm
    | Not(p) -> Not(qelift vars p)
    | And(p,q) -> And(qelift vars p,qelift vars q)
    | Or(p,q) -> Or(qelift vars p,qelift vars q)
    | Imp(p,q) -> Imp(qelift vars p,qelift vars q)
    | Iff(p,q) -> Iff(qelift vars p,qelift vars q)
    | Forall(x,p) -> Not(qelift vars (Exists(x,Not p)))
    | Exists(x,p) ->
         let djs = disjuncts(nfn(qelift (x::vars) p)) in
         list_disj(map (qelim (qfn vars) x) djs)
    | _ -> fm in
  fun fm -> simplify(qelift (fv fm) (miniscope fm));;
```

La función `nnf` es una versión mejorada de la transformación a forma normal disyuntiva. Para ello construiremos a continuación una versión mejorada de la forma normal negativa de una fórmula.

En primer lugar, nos gustaría tener una función para modificar literales, por ejemplo para transformar desigualdades negadas como $\neg(s < t)$ en $t \leq s$. A esta función la denominaremos `lfn`. Esta función será propia de la teoría en que trabajemos.

En segundo lugar, a veces se realizarán divisiones en casos según una propiedad p de las otras variables, obteniendo una fórmula de la forma $(p \wedge q_0) \vee (\neg p \wedge q_1)$. En lugar de negarla y transformarla en forma normal disyuntiva, es menos costoso computacionalmente usar el hecho de que

$$\neg((p \wedge q_0) \vee (\neg p \wedge q_1)) \Leftrightarrow (p \wedge \neg q_0) \vee (\neg p \wedge \neg q_1)$$

Veamos esta equivalencia:

$$\neg((p \wedge q_0) \vee (\neg p \wedge q_1)) \Leftrightarrow (\neg p \vee \neg q_0) \wedge (p \vee \neg q_1) \Leftrightarrow$$
$$\Leftrightarrow (\neg p \wedge p) \vee (\neg p \wedge \neg q_1) \vee (\neg q_0 \wedge p) \vee (\neg q_0 \wedge \neg q_1)$$

Como $\neg p \wedge p$ es una contradicción, obtenemos

$$(\neg p \wedge p) \vee (\neg p \wedge \neg q_1) \vee (\neg q_0 \wedge p) \vee (\neg q_0 \wedge \neg q_1) \Leftrightarrow (\neg p \wedge \neg q_1) \vee (\neg q_0 \wedge p) \vee (\neg q_0 \wedge \neg q_1)$$

Además, bien p o bien $\neg p$ debe ser 'Verdadero':

- p 'Verdadero y $(\neg q_0 \wedge p)$ 'Falso' $\Rightarrow \neg q_0$ 'Falso' $\Rightarrow (\neg q_0 \wedge \neg q_1)$ 'Falso'.
- p 'Falso' y $(\neg p \wedge \neg q_1)$ 'Falso' $\Rightarrow \neg q_1$ 'Falso' $\Rightarrow (\neg q_0 \wedge \neg q_1)$ es 'Falso'.

Recíprocamente, si $(\neg q_0 \wedge \neg q_1)$ es 'Falso', entonces

$$(\neg p \wedge \neg q_1) \vee (\neg q_0 \wedge p) \vee (\neg q_0 \wedge \neg q_1) \Leftrightarrow (\neg p \wedge \neg q_1) \vee (\neg q_0 \wedge p)$$

Tenemos por tanto la equivalencia.

Usando todo esto definiremos la función `cnnf`, simplificando la fórmula tanto al principio como al final:

```
let cnnf lfn =
  let rec cnnf fm =
    match fm with
      And(p,q) -> And(cnnf p,cnnf q)
    | Or(p,q) -> Or(cnnf p,cnnf q)
    | Imp(p,q) -> Or(cnnf(Not p),cnnf q)
    | Iff(p,q) -> Or(And(cnnf p,cnnf q),And(cnnf(Not p),cnnf(Not q)))
    | Not(Not p) -> cnnf p
    | Not(And(p,q)) -> Or(cnnf(Not p),cnnf(Not q))
    | Not(Or(And(p,q),And(p',r))) when p' = negate p ->
        Or(cnnf (And(p,Not q)),cnnf (And(p',Not r)))
    | Not(Or(p,q)) -> And(cnnf(Not p),cnnf(Not q))
    | Not(Imp(p,q)) -> And(cnnf p,cnnf(Not q))
    | Not(Iff(p,q)) -> Or(And(cnnf p,cnnf(Not q)),
                          And(cnnf(Not p),cnnf q))
    | _ -> lfn fm in
  simplify ** cnnf ** simplify;;
```

5.3. Teoría de los órdenes lineales densos

La teoría de los órdenes lineales densos sin extremos (DLO, del inglés 'dense linear orders') está basada en un lenguaje conteniendo el predicado binario '$<$' y la igualdad sin símbolos de función. Puede ser axiomatizada por el siguiente conjunto finito de fórmulas cerradas:

$$\begin{array}{l}
\forall x\ y.\ x = y \vee x < y \vee y < x,\\
\forall x\ y\ z.\ x < y \wedge y < z \Rightarrow x < z,\\
\forall x.\ \neg(x < x),\\
\forall x\ y.\ x < y \Rightarrow \exists z. x < z \wedge z < y,\\
\forall x. \exists y.\ x < y,\\
\forall x. \exists y.\ y < x.
\end{array}$$

Las tres primeras expresan que '$<$' es un orden lineal. La siguiente asegura la densidad, esto es, que entre cada par de elementos hay otro. Las dos últimas fórmulas afirman que no hay extremos.

5.3.1. Igualdad

En lo que sigue usaremos la igualdad como un predicado binario, y es por ello que necesitamos implementarla en Ocaml.

En muchas aplicaciones de la lógica, las ecuaciones juegan un papel central. Podemos definir operaciones sintácticas para probar si una fórmula es una ecuación o para obtener los dos términos que forman una ecuación:

```
let is_eq = function (Atom(R("=",_))) -> true | _ -> false;;

let dest_eq fm =
  match fm with
    Atom(R("=",[s;t])) -> s,t
  | _ -> failwith "dest_eq: not an equation";;
```

5.3.2. Eliminación de cuantificadores para DLO

Esta teoría admite eliminación de cuantificadores, como mostró Langford [7], y daremos un algoritmo explícito para ella. Por los resultados ya vistos, basta considerar una fórmula $\exists x.(l_1(x) \wedge ... \wedge l_n(x))$, donde cada $l_i(x)$ es un literal que contiene a x.

Para implementar el procedimiento usaremos la función `lift_qelim`. Para ello debemos definir:

- `afn`, que denominaremos `afn_dlo`.
- `nfn` como `dnf ** cnnf lfn_dlo`, donde sólo nos queda definir `lfn_dlo`, que es el argumento `lfn` de la función ya definida `cnnf`.
- `qfn`, que denominaremos `dlobasic`, y será el núcleo del procedimiento.

Definimos en primer lugar la función `afn_dlo` como una transformación inicial para permitirnos usar otras relaciones de desigualdad:

- $s \leq t \Leftrightarrow \neg(t < s)$,
- $s \geq t \Leftrightarrow \neg(s < t)$,
- $s > t \Leftrightarrow t < s$.

```
let afn_dlo vars fm =
  match fm with
    Atom(R("<=",[s;t])) -> Not(Atom(R("<",[t;s])))
  | Atom(R(">=",[s;t])) -> Not(Atom(R("<",[s;t])))
  | Atom(R(">",[s;t])) -> Atom(R("<",[t;s]))
  | _ -> fm;;
```

A continuación definiremos la función `lfn_dlo` como una modificación de literales específica para esta teoría, basándonos en las equivalencias:

$$\neg(s < t) \Leftrightarrow s = t \vee t < s$$
$$\neg(s = t) \Leftrightarrow s < t \vee t < s$$

```
let lfn_dlo fm =
  match fm with
    Not(Atom(R("<",[s;t]))) -> Or(Atom(R("=",[s;t])),Atom(R("<",[t;s])))
  | Not(Atom(R("=",[s;t]))) -> Or(Atom(R("<",[s;t])),Atom(R("<",[t;s])))
  | _ -> fm;;
```

Nos centraremos por último en la función `dlobasic`, que es el núcleo del procedimiento. Esta función toma como argumento una fórmula de la forma $\exists x.F$, donde F es una conjunción de literales, y devuelve una fórmula equivalente a la anterior sin cuantificadores.

Ya que no hay símbolos de función en esta teoría podemos suponer que todos los literales de F son átomos. Deben ser de la forma $x < y$ o $x = y$ para ciertas variables x e y.

Cualquier átomo de la forma $x = x$ es trivialmente verdadero y puede ser ignorado; los demás se recogen en una lista `cjs`. Si algunos de estos es una ecuación, entonces, como todos los literales contienen a la variable cuantificada, debe ser de la forma $x = y$ o $y = x$, donde x es la variable existencialmente cuantificada que queremos eliminar e y es otra variable. En este caso podemos obtener una fórmula lógicamente equivalente eliminando el cuantificador y sustituyendo x en las otras subfórmulas; esto sólo refleja las equivalencias lógicas como

$$(\exists x.\ x = y \wedge P(x, y)) \Leftrightarrow P(y, y)$$

Si este paso no puede ser aplicado, entonces todos los átomos deben ser inecuaciones. Si alguno es de la forma $x < x$, entonces él y la fórmula son trivialmente falsos. En otro caso definimos `ls` como el conjunto de los términos s_i que aparecen en inecuaciones $s_i < x$, y como `rs` el conjunto de los términos que aparecen en inecuaciones $x < t_j$. Notemos ahora que en esta teoría:

$$T \Vdash (\exists x.(\bigwedge_i s_i < x) \wedge (\bigwedge_j x < t_j)) \Leftrightarrow (\bigwedge_{i,j} s_i < t_j)$$

Justifiquemos este paso. Si $s_i < x \wedge x < t_j$, usando el axioma de transitividad, concluimos que $s_i < t_j$. Recíprocamente, si $\bigwedge_{i,j} s_i < t_j$, tomando el mayor de los s_i y el menor de los t_j y aplicando el axioma de densidad obtenemos que

$$\bigwedge_{i,j}(s_i < x \wedge x < t_j)$$

Concluimos el resultado gracias al axioma de transitividad.

Si no hay desigualdades de alguno de los dos tipos (`ls` o `rs` son vacíos), la fórmula es equivalente a 'Verdadero', ya que los axiomas de la teoría afirman que no hay extremos. Como `list_conj` devuelve $\top$ para la lista vacía, estos casos ya estarían resueltos en la implementación:

```
let dlobasic fm =
  match fm with
    Exists(x,p) ->
      let cjs = subtract (conjuncts p) [Atom(R("=",[Var x;Var x]))] in
      try let eqn = find is_eq cjs in
          let s,t = dest_eq eqn in
          let y = if s = Var x then t else s in
          list_conj(map (subst (x |=> y)) (subtract cjs [eqn]))
      with Failure _ ->
          if mem (Atom(R("<",[Var x;Var x]))) cjs then False else
          let lefts,rights =
            partition (fun (Atom(R("<",[s;t]))) -> t = Var x) cjs in
          let ls = map (fun (Atom(R("<",[l;_]))) -> l) lefts
          and rs = map (fun (Atom(R("<",[_;r]))) -> r) rights in
          list_conj(allpairs (fun l r -> Atom(R("<",[l;r]))) ls rs)
  | _ -> failwith "dlobasic";;
```

Finalmente usamos la función `lift_qelim` para definir el procedimiento, como ya anticipábamos:

```
let quelim_dlo =
  lift_qelim afn_dlo (dnf ** cnnf lfn_dlo) (fun v -> dlobasic);;
```

5.3.3. Ejemplos

Veamos en primer lugar un ejemplo sencillo:

$$\exists z.(z < x \wedge z < y)$$

Esta fórmula es del tipo $\exists z.F$, donde F es una conjunción de literales, por lo que no hay que aplicarle transformaciones previas. Dichos literales son inecuaciones, ambos de la forma $z < t_j$. Entonces esta fórmula es trivialmente verdadera, ya que los axiomas de la teoría afirman que no hay extremos ($\forall x.\ \exists y.\ y < x$):

```
# quelim_dlo <<exists z. z < x /\ z < y>>;;
- : fol formula = <<true>>
```

La fórmula

$$\exists z.(x < z \wedge z < y)$$

también es del tipo $\exists z.F$, con F es una conjunción de literales. Además dichos literales son también inecuaciones. Por el axioma de transitividad, como $x < z$ y $z < y$ tenemos que:

$$\exists z.(x < z \wedge z < y) \Leftrightarrow x < y,$$

la cual es una fórmula sin cuantificadores.

```
# quelim_dlo <<exists z. x < z /\ z < y>>;;
- : fol formula = <<x < y>>
```

Veamos otro ejemplo. La fórmula

$$\forall x.(x < a \Rightarrow x < b)$$

es un ejemplo de una fórmula a la que hay que realizarle transformaciones previas. Aplicando las Leyes de De Morgan:

$$\forall x.(x < a \Rightarrow x < b) \Leftrightarrow \neg\exists x.\neg(x < a \Rightarrow x < b)$$

Por tanto, la fórmula a la que debemos aplicar el procedimiento es

$$\exists x.\neg(x < a \Rightarrow x < b)$$

Transformando el cuerpo en forma normal negativa:

$$\exists x.\neg(x < a \Rightarrow x < b) \Leftrightarrow \exists x.(x < a \wedge \neg(x < b))$$

Podemos aplicar además una desigualdad propia de esta teoría:

$$\neg(x < b) \Leftrightarrow x = b \vee b < x$$

Entonces:

$$\begin{gathered}\exists x.\neg(x < a \Rightarrow x < b) \Leftrightarrow \exists x.((x < a \wedge x = b) \vee (x < a \wedge b < x)) \Leftrightarrow \\ \Leftrightarrow \exists x.(x < a \wedge x = b) \vee \exists x.(x < a \wedge b < x)\end{gathered}$$

Aplicamos por tanto el procedimiento de eliminación de cuantificadores a ambas subfórmulas existencialmente cuantificadas. En el primer caso, como uno de los literales es la ecuación $x = b$, eliminamos el cuantificador sustituyendo x por b en los demás literales. El segundo caso es similar al ejemplo anterior.

$$\begin{gathered}\exists x.(x < a \wedge x = b) \Leftrightarrow b < a \\ \exists x.(x < a \wedge b < x) \Leftrightarrow b < a\end{gathered}$$

Finalmente hemos conseguido eliminar el cuantificador universal:

$$\forall x.(x < a \Rightarrow x < b) \Leftrightarrow \neg(b < a \vee b < a) \Leftrightarrow \neg(b < a)$$

```
# quelim_dlo <<(forall x. x < a ==> x < b)>>;;
- : fol formula = <<~(b < a \/ b < a)>>
```

Entonces, si aplicamos el procedimiento de eliminación de cuantificadores a la fórmula

$$\forall a\ b.\ ((\forall x. x < a \Rightarrow x < b) \Leftrightarrow a \leq b)$$

debemos obtener que es una tautología, debido al ejemplo anterior.

```
# quelim_dlo <<forall a b. (forall x. x < a ==> x < b) <=> a <= b>>;;
- : fol formula = <<true>>
```

Bibliografía

[1] Instalación de ocaml. https://ocaml.org/docs/install.html. Accedido por última vez el 13-06-2016.

[2] Ocaml. https://ocaml.org/. Accedido por última vez el 13-06-2016.

[3] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.

[4] Amine Chaieb and Tobias Nipkow. Verifying and reflecting quantifier elimination for presburger arithmetic.

[5] Herbert B Enderton. *A mathematical introduction to logic*. Academic press, 2001.

[6] John Harrison. *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009.

[7] Cooper Harold Langford. Some theorems on deducibility. *Annals of Mathematics*, pages 16–40, 1926.

[8] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system: Documentation and user's manual. *INRIA, release*, 4.

[9] Shashi Mohan Srivastava. *A course on mathematical logic*. Springer Science & Business Media, 2013.

Apéndice A

Código

```
(* ========================================================================== *)
(* CÓDIGO INICIAL.                                                            *)
(* ========================================================================== *)

#load "nums.cma";;

if let v = String.sub Sys.ocaml_version 0 4 in v >= "3.10"
then (Topdirs.dir_directory "+camlp5";
      Topdirs.dir_load Format.std_formatter "camlp5o.cma")
else (Topdirs.dir_load Format.std_formatter "camlp4o.cma");;

type dummy_interactive = START_INTERACTIVE | END_INTERACTIVE;;
#use "initialization.ml";;
#use "Quotexpander.ml";;

(* -------------------------------------------------------------------------- *)
(* Algunas funciones básicas.                                                 *)
(* -------------------------------------------------------------------------- *)

let ( ** ) = fun f g x -> f(g x);;

let non p x = not(p x);;

(* -------------------------------------------------------------------------- *)
(* Operaciones sobre listas.                                                  *)
(* -------------------------------------------------------------------------- *)

let hd l =
  match l with
   h::t -> h
  | _ -> failwith "hd";;
```

```
let tl l =
  match l with
   h::t -> t
  | _ -> failwith "tl";;

let rec itlist f l b =
  match l with
    [] -> b
  | (h::t) -> f h (itlist f t b);;

let rec end_itlist f l =
  match l with
        []     -> failwith "end_itlist"
      | [x]    -> x
      | (h::t) -> f h (end_itlist f t);;

let rec forall p l =
  match l with
    [] -> true
  | h::t -> p(h) && forall p t;;

let rec exists p l =
  match l with
    [] -> false
  | h::t -> p(h) || exists p t;;

let partition p l =
  itlist (fun a (yes,no) -> if p a then a::yes,no else yes,a::no) l ([],[]);;

let filter p l = fst(partition p l);;

let length =
  let rec len k l =
    if l = [] then k else len (k + 1) (tl l) in
  fun l -> len 0 l;;

let rec find p l =
  match l with
      [] -> failwith "find"
  | (h::t) -> if p(h) then h else find p t;;

let rec el n l =
  if n = 0 then hd l else el (n - 1) (tl l);;

let map f =
  let rec mapf l =
```

```
    match l with
      [] -> []
    | (x::t) -> let y = f x in y::(mapf t) in
  mapf;;

let rec allpairs f l1 l2 =
  match l1 with
   h1::t1 ->  itlist (fun x a -> f h1 x :: a) l2 (allpairs f t1 l2)
  | [] -> [];;

(* ------------------------------------------------------------------------- *)
(* Aplicación de una función sobre una lista.                                *)
(* ------------------------------------------------------------------------- *)

let rec do_list f l =
  match l with
    [] -> ()
  | h::t -> f(h); do_list f t;;

(* ------------------------------------------------------------------------- *)
(* Combinación de listas ordenadas.                                          *)
(* ------------------------------------------------------------------------- *)

let rec merge ord l1 l2 =
  match l1 with
    [] -> l2
  | h1::t1 -> match l2 with
                [] -> l1
              | h2::t2 -> if ord h1 h2 then h1::(merge ord t1 l2)
                else h2::(merge ord l1 t2);;

let sort ord =
  let rec mergepairs l1 l2 =
    match (l1,l2) with
        ([s],[]) -> s
      | (l,[]) -> mergepairs [] l
      | (l,[s1]) -> mergepairs (s1::l) []
      | (l,(s1::s2::ss)) -> mergepairs ((merge ord s1 s2)::l) ss in
  fun l -> if l = [] then [] else mergepairs [] (map (fun x -> [x]) l);;

(* ------------------------------------------------------------------------- *)
(* Eliminar repeticiones.                                                    *)
(* ------------------------------------------------------------------------- *)

let rec uniq l =
  match l with
```

```
    x::(y::_ as t) -> let t' = uniq t in
                      if Pervasives.compare x y = 0 then t' else
                      if t'==t then l else x::t'
  | _ -> l;;

(* ------------------------------------------------------------------------- *)
(* Maximizar o minimizar funciones.                                          *)
(* ------------------------------------------------------------------------- *)

let optimize ord f l =
  fst(end_itlist (fun (x,y as p) (x',y' as p') -> if ord y y' then p else p')
        (map (fun x -> x,f x) l));;

let maximize f l = optimize (>) f l and minimize f l = optimize (<) f l;;

(* ------------------------------------------------------------------------- *)
(* Operaciones sobre conjuntos en listas ordenadas.                          *)
(* ------------------------------------------------------------------------- *)

let setify =
  let rec canonical lis =
     match lis with
       x::(y::_ as rest) -> Pervasives.compare x y < 0 && canonical rest
     | _ -> true in
  fun l -> if canonical l then l
    else uniq (sort (fun x y -> Pervasives.compare x y <= 0) l);;

let union =
  let rec union l1 l2 =
    match (l1,l2) with
        ([],l2) -> l2
      | (l1,[]) -> l1
      | ((h1::t1 as l1),(h2::t2 as l2)) ->
          if h1 = h2 then h1::(union t1 t2)
          else if h1 < h2 then h1::(union t1 l2)
          else h2::(union l1 t2) in
  fun s1 s2 -> union (setify s1) (setify s2);;

let intersect =
  let rec intersect l1 l2 =
    match (l1,l2) with
        ([],l2) -> []
      | (l1,[]) -> []
      | ((h1::t1 as l1),(h2::t2 as l2)) ->
          if h1 = h2 then h1::(intersect t1 t2)
          else if h1 < h2 then intersect t1 l2
```

```
          else intersect l1 t2 in
  fun s1 s2 -> intersect (setify s1) (setify s2);;

let subtract =
  let rec subtract l1 l2 =
    match (l1,l2) with
        ([],l2) -> []
      | (l1,[]) -> l1
      | ((h1::t1 as l1),(h2::t2 as l2)) ->
          if h1 = h2 then subtract t1 t2
          else if h1 < h2 then h1::(subtract t1 l2)
          else subtract l1 t2 in
  fun s1 s2 -> subtract (setify s1) (setify s2);;

let subset,psubset =
  let rec subset l1 l2 =
    match (l1,l2) with
        ([],l2) -> true
      | (l1,[]) -> false
      | (h1::t1,h2::t2) ->
          if h1 = h2 then subset t1 t2
          else if h1 < h2 then false
          else subset l1 t2
  and psubset l1 l2 =
    match (l1,l2) with
        (l1,[]) -> false
      | ([],l2) -> true
      | (h1::t1,h2::t2) ->
          if h1 = h2 then psubset t1 t2
          else if h1 < h2 then false
          else subset l1 t2 in
  (fun s1 s2 -> subset (setify s1) (setify s2)),
  (fun s1 s2 -> psubset (setify s1) (setify s2));;

let insert x s = union [x] s;;

let image f s = setify (map f s);;

(* ------------------------------------------------------------------------- *)
(* Unión de una familia de conjuntos.                                        *)
(* ------------------------------------------------------------------------- *)

let unions s = setify(itlist (@) s []);;

(* ------------------------------------------------------------------------- *)
(* Pertenencia a una lista.                                                  *)
```

```
(* ------------------------------------------------------------------------ *)

let rec mem x lis =
  match lis with
    [] -> false
  | (h::t) -> Pervasives.compare x h = 0 || mem x t;;

(* ------------------------------------------------------------------------ *)
(* Explosión de cadenas.                                                    *)
(* ------------------------------------------------------------------------ *)

let explode s =
  let rec exap n l =
     if n < 0 then l else
      exap (n - 1) ((String.sub s n 1)::l) in
  exap (String.length s - 1) [];;

(* ------------------------------------------------------------------------ *)
(* Árboles.                                                                 *)
(* ------------------------------------------------------------------------ *)

type ('a,'b)func =
   Empty
 | Leaf of int * ('a*'b)list
 | Branch of int * int * ('a,'b)func * ('a,'b)func;;

(* ------------------------------------------------------------------------ *)
(* Función indefinida.                                                      *)
(* ------------------------------------------------------------------------ *)

let undefined = Empty;;

(* ------------------------------------------------------------------------ *)
(* Operaciones fold para árboles.                                           *)
(* ------------------------------------------------------------------------ *)

let foldl =
  let rec foldl_list f a l =
    match l with
      [] -> a
    | (x,y)::t -> foldl_list f (f a x y) t in
  let rec foldl f a t =
    match t with
      Empty -> a
    | Leaf(h,l) -> foldl_list f a l
    | Branch(p,b,l,r) -> foldl f (foldl f a l) r in
```

```
  foldl;;

(* ------------------------------------------------------------------------ *)
(* Grafo de una función.                                                    *)
(* ------------------------------------------------------------------------ *)

let graph f = setify (foldl (fun a x y -> (x,y)::a) [] f);;

(* ------------------------------------------------------------------------ *)
(* Aplicación.                                                              *)
(* ------------------------------------------------------------------------ *)

let applyd =
  let rec apply_listd l d x =
    match l with
      (a,b)::t -> let c = Pervasives.compare x a in
                  if c = 0 then b else if c > 0 then apply_listd t d x else d x
    | [] -> d x in
  fun f d x ->
    let k = Hashtbl.hash x in
    let rec look t =
      match t with
        Leaf(h,l) when h = k -> apply_listd l d x
      | Branch(p,b,l,r) when (k lxor p) land (b - 1) = 0
                -> look (if k land b = 0 then l else r)
      | _ -> d x in
    look f;;

let apply f = applyd f (fun x -> failwith "apply");;

let tryapplyd f a d = applyd f (fun x -> d) a;;

(* ------------------------------------------------------------------------ *)
(* Indefinido.                                                              *)
(* ------------------------------------------------------------------------ *)

let undefine =
  let rec undefine_list x l =
    match l with
      (a,b as ab)::t ->
          let c = Pervasives.compare x a in
          if c = 0 then t
          else if c < 0 then l else
          let t' = undefine_list x t in
          if t' == t then l else ab::t'
    | [] -> [] in
```

```
  fun x ->
    let k = Hashtbl.hash x in
    let rec und t =
      match t with
        Leaf(h,l) when h = k ->
          let l' = undefine_list x l in
          if l' == l then t
          else if l' = [] then Empty
          else Leaf(h,l')
      | Branch(p,b,l,r) when k land (b - 1) = p ->
          if k land b = 0 then
            let l' = und l in
            if l' == l then t
            else (match l' with Empty -> r | _ -> Branch(p,b,l',r))
          else
            let r' = und r in
            if r' == r then t
            else (match r' with Empty -> l | _ -> Branch(p,b,l,r'))
      | _ -> t in
    und;;

(* ------------------------------------------------------------------------ *)
(* Redfinición y combinación.                                               *)
(* ------------------------------------------------------------------------ *)

let (|->),combine =
  let newbranch p1 t1 p2 t2 =
    let zp = p1 lxor p2 in
    let b = zp land (-zp) in
    let p = p1 land (b - 1) in
    if p1 land b = 0 then Branch(p,b,t1,t2)
    else Branch(p,b,t2,t1) in
  let rec define_list (x,y as xy) l =
    match l with
      (a,b as ab)::t ->
          let c = Pervasives.compare x a in
          if c = 0 then xy::t
          else if c < 0 then xy::l
          else ab::(define_list xy t)
    | [] -> [xy]
  and combine_list op z l1 l2 =
    match (l1,l2) with
      [],_ -> l2
    | _,[] -> l1
    | ((x1,y1 as xy1)::t1,(x2,y2 as xy2)::t2) ->
          let c = Pervasives.compare x1 x2 in
```

```
        if c < 0 then xy1::(combine_list op z t1 l2)
        else if c > 0 then xy2::(combine_list op z l1 t2) else
        let y = op y1 y2 and l = combine_list op z t1 t2 in
        if z(y) then l else (x1,y)::l in
let (|->) x y =
  let k = Hashtbl.hash x in
  let rec upd t =
    match t with
      Empty -> Leaf (k,[x,y])
    | Leaf(h,l) ->
         if h = k then Leaf(h,define_list (x,y) l)
         else newbranch h t k (Leaf(k,[x,y]))
    | Branch(p,b,l,r) ->
        if k land (b - 1) <> p then newbranch p t k (Leaf(k,[x,y]))
        else if k land b = 0 then Branch(p,b,upd l,r)
        else Branch(p,b,l,upd r) in
  upd in
let rec combine op z t1 t2 =
  match (t1,t2) with
    Empty,_ -> t2
  | _,Empty -> t1
  | Leaf(h1,l1),Leaf(h2,l2) ->
        if h1 = h2 then
          let l = combine_list op z l1 l2 in
          if l = [] then Empty else Leaf(h1,l)
        else newbranch h1 t1 h2 t2
  | (Leaf(k,lis) as lf),(Branch(p,b,l,r) as br) ->
        if k land (b - 1) = p then
          if k land b = 0 then
            (match combine op z lf l with
               Empty -> r | l' -> Branch(p,b,l',r))
          else
            (match combine op z lf r with
               Empty -> l | r' -> Branch(p,b,l,r'))
        else
          newbranch k lf p br
  | (Branch(p,b,l,r) as br),(Leaf(k,lis) as lf) ->
        if k land (b - 1) = p then
          if k land b = 0 then
            (match combine op z l lf with
              Empty -> r | l' -> Branch(p,b,l',r))
          else
            (match combine op z r lf with
               Empty -> l | r' -> Branch(p,b,l,r'))
        else
          newbranch p br k lf
```

```
    | Branch(p1,b1,l1,r1),Branch(p2,b2,l2,r2) ->
          if b1 < b2 then
            if p2 land (b1 - 1) <> p1 then newbranch p1 t1 p2 t2
            else if p2 land b1 = 0 then
              (match combine op z l1 t2 with
                 Empty -> r1 | l -> Branch(p1,b1,l,r1))
            else
              (match combine op z r1 t2 with
                 Empty -> l1 | r -> Branch(p1,b1,l1,r))
          else if b2 < b1 then
            if p1 land (b2 - 1) <> p2 then newbranch p1 t1 p2 t2
            else if p1 land b2 = 0 then
              (match combine op z t1 l2 with
                 Empty -> r2 | l -> Branch(p2,b2,l,r2))
            else
              (match combine op z t1 r2 with
                 Empty -> l2 | r -> Branch(p2,b2,l2,r))
          else if p1 = p2 then
           (match (combine op z l1 l2,combine op z r1 r2) with
              (Empty,r) -> r | (l,Empty) -> l | (l,r) -> Branch(p1,b1,l,r))
          else
            newbranch p1 t1 p2 t2 in
  (|->),combine;;

let (|=>) = fun x y -> (x |-> y) undefined;;

(* ------------------------------------------------------------------------- *)
(* Análisis léxico.                                                          *)
(* ------------------------------------------------------------------------- *)

let matches s = let chars = explode s in fun c -> mem c chars;;

let space = matches " \t\n\r"
and punctuation = matches "()[]{},"
and symbolic = matches "~`!@#$%^&*-+=|\\:;<>.?/"
and numeric = matches "0123456789"
and alphanumeric = matches
  "abcdefghijklmnopqrstuvwxyz_'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";;

let rec lexwhile prop inp =
  match inp with
    c::cs when prop c -> let tok,rest = lexwhile prop cs in c^tok,rest
  | _ -> "",inp;;

let rec lex inp =
  match snd(lexwhile space inp) with
```

```
    [] -> []
  | c::cs -> let prop = if alphanumeric(c) then alphanumeric
                        else if symbolic(c) then symbolic
                        else fun c -> false in
             let toktl,rest = lexwhile prop cs in
             (c^toktl)::lex rest;;

(* ------------------------------------------------------------------------- *)
(* Parser.                                                                   *)
(* ------------------------------------------------------------------------- *)

let make_parser pfn s =
  let expr,rest = pfn (lex(explode s)) in
  if rest = [] then expr else failwith "Unparsed input";;


(* ========================================================================= *)
(* Sintaxis de la lógica proposicional.                                      *)
(* ========================================================================= *)

type ('a)formula = False
                 | True
                 | Atom of 'a
                 | Not of ('a)formula
                 | And of ('a)formula * ('a)formula
                 | Or of ('a)formula * ('a)formula
                 | Imp of ('a)formula * ('a)formula
                 | Iff of ('a)formula * ('a)formula
                 | Forall of string * ('a)formula
                 | Exists of string * ('a)formula;;

(* ------------------------------------------------------------------------- *)
(* Parsing general.                                                          *)
(* ------------------------------------------------------------------------- *)

let rec parse_ginfix opsym opupdate sof subparser inp =
  let e1,inp1 = subparser inp in
  if inp1 <> [] && hd inp1 = opsym then
     parse_ginfix opsym opupdate (opupdate sof e1) subparser (tl inp1)
  else sof e1,inp1;;

let parse_left_infix opsym opcon =
  parse_ginfix opsym (fun f e1 e2 -> opcon(f e1,e2)) (fun x -> x);;

let parse_right_infix opsym opcon =
  parse_ginfix opsym (fun f e1 e2 -> f(opcon(e1,e2))) (fun x -> x);;
```

```
let parse_list opsym =
  parse_ginfix opsym (fun f e1 e2 -> (f e1)@[e2]) (fun x -> [x]);;


let papply f (ast,rest) = (f ast,rest);;

let nextin inp tok = inp <> [] && hd inp = tok;;

let parse_bracketed subparser cbra inp =
  let ast,rest = subparser inp in
  if nextin rest cbra then ast,tl rest
  else failwith "Closing bracket expected";;

(* ------------------------------------------------------------------------- *)
(* Parsing de fórmulas.                                                      *)
(* ------------------------------------------------------------------------- *)

let rec parse_atomic_formula (ifn,afn) vs inp =
  match inp with
    [] -> failwith "formula expected"
  | "false"::rest -> False,rest
  | "true"::rest -> True,rest
  | "("::rest -> (try ifn vs inp with Failure _ ->
                  parse_bracketed (parse_formula (ifn,afn) vs) ")" rest)
  | "~"::rest -> papply (fun p -> Not p)
                        (parse_atomic_formula (ifn,afn) vs rest)
  | "forall"::x::rest ->
        parse_quant (ifn,afn) (x::vs) (fun (x,p) -> Forall(x,p)) x rest
  | "exists"::x::rest ->
        parse_quant (ifn,afn) (x::vs) (fun (x,p) -> Exists(x,p)) x rest
  | _ -> afn vs inp

and parse_quant (ifn,afn) vs qcon x inp =
   match inp with
     [] -> failwith "Body of quantified term expected"
   | y::rest ->
        papply (fun fm -> qcon(x,fm))
               (if y = "." then parse_formula (ifn,afn) vs rest
                else parse_quant (ifn,afn) (y::vs) qcon y rest)

and parse_formula (ifn,afn) vs inp =
   parse_right_infix "<=>" (fun (p,q) -> Iff(p,q))
     (parse_right_infix "==>" (fun (p,q) -> Imp(p,q))
         (parse_right_infix "\\/" (fun (p,q) -> Or(p,q))
             (parse_right_infix "/\\" (fun (p,q) -> And(p,q))
```

```
                (parse_atomic_formula (ifn,afn) vs)))) inp;;

(* ------------------------------------------------------------------------- *)
(* Printing de fórmulas.                                                     *)
(* ------------------------------------------------------------------------- *)

let bracket p n f x y =
  (if p then print_string "(" else ());
  open_box n; f x y; close_box();
  (if p then print_string ")" else ());;

let rec strip_quant fm =
  match fm with
    Forall(x,(Forall(y,p) as yp)) | Exists(x,(Exists(y,p) as yp)) ->
        let xs,q = strip_quant yp in x::xs,q
  |  Forall(x,p) | Exists(x,p) -> [x],p
  | _ -> [],fm;;

let print_formula pfn =
  let rec print_formula pr fm =
    match fm with
      False -> print_string "false"
    | True -> print_string "true"
    | Atom(pargs) -> pfn pr pargs
    | Not(p) -> bracket (pr > 10) 1 (print_prefix 10) "~" p
    | And(p,q) -> bracket (pr > 8) 0 (print_infix 8 "/\\") p q
    | Or(p,q) ->  bracket (pr > 6) 0 (print_infix  6 "\\/") p q
    | Imp(p,q) ->  bracket (pr > 4) 0 (print_infix 4 "==>") p q
    | Iff(p,q) ->  bracket (pr > 2) 0 (print_infix 2 "<=>") p q
    | Forall(x,p) -> bracket (pr > 0) 2 print_qnt "forall" (strip_quant fm)
    | Exists(x,p) -> bracket (pr > 0) 2 print_qnt "exists" (strip_quant fm)
  and print_qnt qname (bvs,bod) =
    print_string qname;
    do_list (fun v -> print_string " "; print_string v) bvs;
    print_string "."; print_space(); open_box 0;
    print_formula 0 bod;
    close_box()
  and print_prefix newpr sym p =
   print_string sym; print_formula (newpr+1) p
  and print_infix newpr sym p q =
    print_formula (newpr+1) p;
    print_string(" "^sym); print_space();
    print_formula newpr q in
  print_formula 0;;

let print_qformula pfn fm =
```

```
  open_box 0; print_string "<<";
  open_box 0; print_formula pfn fm; close_box();
  print_string ">>"; close_box();;

(* ------------------------------------------------------------------------- *)
(* Variables proposicionales.                                                *)
(* ------------------------------------------------------------------------- *)

type prop = P of string;;

let pname(P s) = s;;

(* ------------------------------------------------------------------------- *)
(* Parsing y printing de fórmulas proposicionales.                           *)
(* ------------------------------------------------------------------------- *)

let parse_propvar vs inp =
  match inp with
    p::oinp when p <> "(" -> Atom(P(p)),oinp
  | _ -> failwith "parse_propvar";;

let parse_prop_formula = make_parser
  (parse_formula ((fun _ _ -> failwith ""),parse_propvar) []);;

let default_parser = parse_prop_formula;;

let print_propvar prec p = print_string(pname p);;

let print_prop_formula = print_qformula print_propvar;;

#install_printer print_prop_formula;;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
let fm = <<p ==> q <=> r /\ s \/ (t <=>  ~  ~u /\ v)>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Constructores como funciones.                                             *)
(* ------------------------------------------------------------------------- *)

let mk_and p q = And(p,q) and mk_or p q = Or(p,q)
and mk_imp p q = Imp(p,q) and mk_iff p q = Iff(p,q)
```

```
and mk_forall x p = Forall(x,p) and mk_exists x p = Exists(x,p);;

(* ------------------------------------------------------------------------- *)
(* Destructores.                                                             *)
(* ------------------------------------------------------------------------- *)

let dest_iff fm =
  match fm with Iff(p,q) -> (p,q) | _ -> failwith "dest_iff";;

let dest_and fm =
  match fm with And(p,q) -> (p,q) | _ -> failwith "dest_and";;

let rec conjuncts fm =
  match fm with And(p,q) -> conjuncts p @ conjuncts q | _ -> [fm];;

let dest_or fm =
  match fm with Or(p,q) -> (p,q) | _ -> failwith "dest_or";;

let rec disjuncts fm =
  match fm with Or(p,q) -> disjuncts p @ disjuncts q | _ -> [fm];;

let dest_imp fm =
  match fm with Imp(p,q) -> (p,q) | _ -> failwith "dest_imp";;

let antecedent fm = fst(dest_imp fm);;
let consequent fm = snd(dest_imp fm);;

(* ------------------------------------------------------------------------- *)
(* Aplicar una función a los átomos.                                         *)
(* ------------------------------------------------------------------------- *)

let rec onatoms f fm =
  match fm with
    Atom a -> f a
  | Not(p) -> Not(onatoms f p)
  | And(p,q) -> And(onatoms f p,onatoms f q)
  | Or(p,q) -> Or(onatoms f p,onatoms f q)
  | Imp(p,q) -> Imp(onatoms f p,onatoms f q)
  | Iff(p,q) -> Iff(onatoms f p,onatoms f q)
  | Forall(x,p) -> Forall(x,onatoms f p)
  | Exists(x,p) -> Exists(x,onatoms f p)
  | _ -> fm;;

let rec overatoms f fm b =
  match fm with
    Atom(a) -> f a b
```

```
  | Not(p) -> overatoms f p b
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) ->
        overatoms f p (overatoms f q b)
  | Forall(x,p) | Exists(x,p) -> overatoms f p b
  | _ -> b;;

let atom_union f fm = setify (overatoms (fun h t -> f(h)@t) fm []);;


(* ========================================================================= *)
(* Semántica de la lógica proposicional.                                     *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Interpretación de fórmulas.                                               *)
(* ------------------------------------------------------------------------- *)

let rec eval fm v =
  match fm with
    False -> false
  | True -> true
  | Atom(x) -> v(x)
  | Not(p) -> not(eval p v)
  | And(p,q) -> (eval p v) && (eval q v)
  | Or(p,q) -> (eval p v) || (eval q v)
  | Imp(p,q) -> not(eval p v) || (eval q v)
  | Iff(p,q) -> (eval p v) = (eval q v);;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
eval <<p /\ q ==> q /\ r>>
  (function P"p" -> true | P"q" -> false | P"r" -> true);;

eval <<p /\ q ==> q /\ r>>
(function P"p" -> true | P"q" -> true | P"r" -> false);;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Devolver el conjunto de fórmulas proposicionales.                         *)
(* ------------------------------------------------------------------------- *)

let atoms fm = atom_union (fun a -> [a]) fm;;
```

```
(* ------------------------------------------------------------------------- *)
(* Tablas de verdad.                                                         *)
(* ------------------------------------------------------------------------- *)

let rec onallvaluations subfn v ats =
  match ats with
    [] -> subfn v
  | p::ps -> let v' t q = if q = p then t else v(q) in
             onallvaluations subfn (v' false) ps &&
             onallvaluations subfn (v' true) ps;;

let print_truthtable fm =
  let ats = atoms fm in
  let width = itlist (max ** String.length ** pname) ats 5 + 1 in
  let fixw s = s^String.make(width - String.length s) ' ' in
  let truthstring p = fixw (if p then "true" else "false") in
  let mk_row v =
     let lis = map (fun x -> truthstring(v x)) ats
     and ans = truthstring(eval fm v) in
     print_string(itlist (^) lis ("| "^ans)); print_newline(); true in
  let separator = String.make (width * length ats + 9) '-' in
  print_string(itlist (fun s t -> fixw(pname s) ^ t) ats "| formula");
  print_newline(); print_string separator; print_newline();
  let _ = onallvaluations mk_row (fun x -> false) ats in
  print_string separator; print_newline();;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
print_truthtable <<p /\ q ==> q /\ r>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Reconociendo tautologías.                                                 *)
(* ------------------------------------------------------------------------- *)

let tautology fm =
  onallvaluations (eval fm) (fun s -> false) (atoms fm);;

(* ------------------------------------------------------------------------- *)
(* Conceptos relacionados.                                                   *)
(* ------------------------------------------------------------------------- *)

let unsatisfiable fm = tautology(Not fm);;
```

```
let satisfiable fm = not(unsatisfiable fm);;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
tautology <<p \/  ~p>>;;

tautology <<p /\  ~p>>;;

unsatisfiable <<p /\  ~p>>;;

tautology <<p \/ q ==> q \/ (p <=> q)>>;;

satisfiable <<p \/ q ==> q \/ (p <=> q)>>;;
END_INTERACTIVE;;

(* ========================================================================= *)
(* Simplificación y forma normal negativa.                                   *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Simplificación.                                                           *)
(* ------------------------------------------------------------------------- *)

let psimplify1 fm =
  match fm with
    Not False -> True
  | Not True -> False
  | Not(Not p) -> p
  | And(p,False) | And(False,p) -> False
  | And(p,True) | And(True,p) -> p
  | Or(p,False) | Or(False,p) -> p
  | Or(p,True) | Or(True,p) -> True
  | Imp(False,p) | Imp(p,True) -> True
  | Imp(True,p) -> p
  | Imp(p,False) -> Not p
  | Iff(p,True) | Iff(True,p) -> p
  | Iff(p,False) | Iff(False,p) -> Not p
  | _ -> fm;;

let rec psimplify fm =
  match fm with
  | Not p -> psimplify1 (Not(psimplify p))
```

```
  | And(p,q) -> psimplify1 (And(psimplify p,psimplify q))
  | Or(p,q) -> psimplify1 (Or(psimplify p,psimplify q))
  | Imp(p,q) -> psimplify1 (Imp(psimplify p,psimplify q))
  | Iff(p,q) -> psimplify1 (Iff(psimplify p,psimplify q))
  | _ -> fm;;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
psimplify <<(true ==> (x <=> false)) ==>  ~(y \/ false /\ z)>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Algunas operaciones de literales.                                         *)
(* ------------------------------------------------------------------------- *)

let negative = function (Not p) -> true | _ -> false;;

let positive lit = not(negative lit);;

let negate = function (Not p) -> p | p -> Not p;;

(* ------------------------------------------------------------------------- *)
(* Forma normal negativa.                                                    *)
(* ------------------------------------------------------------------------- *)

let rec nnf fm =
  match fm with
  | And(p,q) -> And(nnf p,nnf q)
  | Or(p,q) -> Or(nnf p,nnf q)
  | Imp(p,q) -> Or(nnf(Not p),nnf q)
  | Iff(p,q) -> Or(And(nnf p,nnf q),And(nnf(Not p),nnf(Not q)))
  | Not(Not p) -> nnf p
  | Not(And(p,q)) -> Or(nnf(Not p),nnf(Not q))
  | Not(Or(p,q)) -> And(nnf(Not p),nnf(Not q))
  | Not(Imp(p,q)) -> And(nnf p,nnf(Not q))
  | Not(Iff(p,q)) -> Or(And(nnf p,nnf(Not q)),And(nnf(Not p),nnf q))
  | _ -> fm;;

let nnf fm = nnf(psimplify fm);;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)
```

```
START_INTERACTIVE;;
let fm = <<(p <=> q) <=> ~(r ==> s)>>;;

let fm' = nnf fm;;

tautology(Iff(fm,fm'));;
   END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Forma normal negativa con bicondicionales.                                *)
(* ------------------------------------------------------------------------- *)

let rec nenf fm =
  match fm with
    Not(Not p) -> nenf p
  | Not(And(p,q)) -> Or(nenf(Not p),nenf(Not q))
  | Not(Or(p,q)) -> And(nenf(Not p),nenf(Not q))
  | Not(Imp(p,q)) -> And(nenf p,nenf(Not q))
  | Not(Iff(p,q)) -> Iff(nenf p,nenf(Not q))
  | And(p,q) -> And(nenf p,nenf q)
  | Or(p,q) -> Or(nenf p,nenf q)
  | Imp(p,q) -> Or(nenf(Not p),nenf q)
  | Iff(p,q) -> Iff(nenf p,nenf q)
  | _ -> fm;;

let nenf fm = nenf(psimplify fm);;

(* ========================================================================= *)
(* Formas normales disyuntivas y conjuntivas.                                *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Forma normal disyuntiva.                                *)
(* ------------------------------------------------------------------------- *)

let list_conj l = if l = [] then True else end_itlist mk_and l;;

let list_disj l = if l = [] then False else end_itlist mk_or l;;

let rec distrib fm =
  match fm with
    And(p,(Or(q,r))) -> Or(distrib(And(p,q)),distrib(And(p,r)))
  | And(Or(p,q),r) -> Or(distrib(And(p,r)),distrib(And(q,r)))
  | _ -> fm;;
```

```
let rec rawdnf fm =
  match fm with
    And(p,q) -> distrib(And(rawdnf p,rawdnf q))
  | Or(p,q) -> Or(rawdnf p,rawdnf q)
  | _ -> fm;;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
rawdnf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Representación como listas.                                               *)
(* ------------------------------------------------------------------------- *)

let distrib s1 s2 = setify(allpairs union s1 s2);;

let rec purednf fm =
  match fm with
    And(p,q) -> distrib (purednf p) (purednf q)
  | Or(p,q) -> union (purednf p) (purednf q)
  | _ -> [[fm]];;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
purednf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
   END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Simplificación.                                                           *)
(* ------------------------------------------------------------------------- *)

let trivial lits =
  let pos,neg = partition positive lits in
  intersect pos (image negate neg) <> [];;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)
```

```
START_INTERACTIVE;;
filter (non trivial) (purednf <<(p \/ q /\ r) /\ ( ~p \/ ~r)>>);;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Simplificación.                                                           *)
(* ------------------------------------------------------------------------- *)

let psubst subfn = onatoms (fun p -> tryapplyd subfn p (Atom p));;

let simpdnf fm =
  if fm = False then [] else if fm = True then [[]] else
  let djs = filter (non trivial) (purednf(nnf fm)) in
  filter (fun d -> not(exists (fun d' -> psubset d' d) djs)) djs;;

(* ------------------------------------------------------------------------- *)
(* Representación como fórmulas.                                             *)
(* ------------------------------------------------------------------------- *)

let dnf fm = list_disj(map list_conj (simpdnf fm));;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
let fm = <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
dnf fm;;
tautology(Iff(fm,dnf fm));;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Forma normal conjuntiva usando DNF.                                       *)
(* ------------------------------------------------------------------------- *)

let purecnf fm = image (image negate) (purednf(nnf(Not fm)));;

let simpcnf fm =
  if fm = False then [[]] else if fm = True then [] else
  let cjs = filter (non trivial) (purecnf fm) in
  filter (fun c -> not(exists (fun c' -> psubset c' c) cjs)) cjs;;

let cnf fm = list_conj(map list_disj (simpcnf fm));;

(* ------------------------------------------------------------------------- *)
(* Example.                                                                  *)
```

```
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
let fm = <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
cnf fm;;
tautology(Iff(fm,cnf fm));;
   END_INTERACTIVE;;


(* ========================================================================= *)
(* FNC usando abreviaciones.                                                 *)
(* ========================================================================= *)


let mkprop n = Atom(P("p_"^(string_of_num n))),n +/ Int 1;;

(* ------------------------------------------------------------------------- *)
(* Núcleo del procedimiento.                                                 *)
(* ------------------------------------------------------------------------- *)

let rec maincnf (fm,defs,n as trip) =
  match fm with
    And(p,q) -> defstep mk_and (p,q) trip
  | Or(p,q) -> defstep mk_or (p,q) trip
  | Iff(p,q) -> defstep mk_iff (p,q) trip
  | _ -> trip

and defstep op (p,q) (fm,defs,n) =
  let fm1,defs1,n1 = maincnf (p,defs,n) in
  let fm2,defs2,n2 = maincnf (q,defs1,n1) in
  let fm' = op fm1 fm2 in
  try (fst(apply defs2 fm'),defs2,n2) with Failure _ ->
  let v,n3 = mkprop n2 in (v,(fm'|->(v,Iff(v,fm'))) defs2,n3);;

(* ------------------------------------------------------------------------- *)
(* Función sobre el índice.                                                  *)
(* ------------------------------------------------------------------------- *)

let max_varindex pfx =
  let m = String.length pfx in
  fun s n ->
    let l = String.length s in
    if l <= m or String.sub s 0 m <> pfx then n else
    let s' = String.sub s m (l - m) in
    if forall numeric (explode s') then max_num n (num_of_string s')
    else n;;
```

```
(* ------------------------------------------------------------------------- *)
(* Procedimiento completo.                                                   *)
(* ------------------------------------------------------------------------- *)

let mk_defcnf fn fm =
  let fm' = nenf fm in
  let n = Int 1 +/ overatoms (max_varindex "p_" ** pname) fm' (Int 0) in
  let (fm'',defs,_) = fn (fm',undefined,n) in
  let deflist = map (snd ** snd) (graph defs) in
  unions(simpcnf fm'' :: map simpcnf deflist);;

let defcnf fm = list_conj(map list_disj(mk_defcnf maincnf fm));;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
defcnf <<(p \/ (q /\ ~r)) /\ s>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Optimización del procedimiento.                                           *)
(* ------------------------------------------------------------------------- *)

let subcnf sfn op (p,q) (fm,defs,n) =
  let fm1,defs1,n1 = sfn(p,defs,n) in
  let fm2,defs2,n2 = sfn(q,defs1,n1) in (op fm1 fm2,defs2,n2);;

let rec orcnf (fm,defs,n as trip) =
  match fm with
    Or(p,q) -> subcnf orcnf mk_or (p,q) trip
  | _ -> maincnf trip;;

let rec andcnf (fm,defs,n as trip) =
  match fm with
    And(p,q) -> subcnf andcnf mk_and (p,q) trip
  | _ -> orcnf trip;;

let defcnfs fm = mk_defcnf andcnf fm;;

let defcnf fm = list_conj (map list_disj (defcnfs fm));;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
```

```
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
defcnf <<(p \/ (q /\ ~r)) /\ s>>;;
END_INTERACTIVE;;


(* ========================================================================= *)
(* Los procedimientos DP y DPLL.                                             *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* El procedimiento DP.                                                      *)
(* ------------------------------------------------------------------------- *)

let one_literal_rule clauses =
  let u = hd (find (fun cl -> length cl = 1) clauses) in
  let u' = negate u in
  let clauses1 = filter (fun cl -> not (mem u cl)) clauses in
  image (fun cl -> subtract cl [u']) clauses1;;

let affirmative_negative_rule clauses =
  let neg',pos = partition negative (unions clauses) in
  let neg = image negate neg' in
  let pos_only = subtract pos neg and neg_only = subtract neg pos in
  let pure = union pos_only (image negate neg_only) in
  if pure = [] then failwith "affirmative_negative_rule" else
    filter (fun cl -> intersect cl pure = []) clauses;;

let resolve_on p clauses =
  let p' = negate p and pos,notpos = partition (mem p) clauses in
  let neg,other = partition (mem p') notpos in
  let pos' = image (filter (fun l -> l <> p)) pos
  and neg' = image (filter (fun l -> l <> p')) neg in
  let res0 = allpairs union pos' neg' in
  union other (filter (non trivial) res0);;

let resolution_blowup cls l =
  let m = length(filter (mem l) cls)
  and n = length(filter (mem (negate l)) cls) in
  m * n - m - n;;

let resolution_rule clauses =
  let pvs = filter positive (unions clauses) in
  let p = minimize (resolution_blowup clauses) pvs in
  resolve_on p clauses;;
```

```
(* ------------------------------------------------------------------------- *)
(* Procedimiento completo.                                                   *)
(* ------------------------------------------------------------------------- *)

let rec dp clauses =
  if clauses = [] then true else if mem [] clauses then false else
  try dp (one_literal_rule clauses) with Failure _ ->
  try dp (affirmative_negative_rule clauses) with Failure _ ->
    dp(resolution_rule clauses);;

(* ------------------------------------------------------------------------- *)
(* Satisfacibilidad y tautología usando DP.                                  *)
(* ------------------------------------------------------------------------- *)

let dpsat fm = dp(defcnfs fm);;

let dptaut fm = not(dpsat(Not fm));;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
tautology <<(p \/ (q /\  ~r)) /\ s>>;;

dptaut <<(p \/ (q /\  ~r)) /\ s>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Procedimiento DPLL.                                                       *)
(* ------------------------------------------------------------------------- *)

let posneg_count cls l =
  let m = length(filter (mem l) cls)
  and n = length(filter (mem (negate l)) cls) in
  m + n;;

let rec dpll clauses =
  if clauses = [] then true else if mem [] clauses then false else
  try dpll(one_literal_rule clauses) with Failure _ ->
  try dpll(affirmative_negative_rule clauses) with Failure _ ->
  let pvs = filter positive (unions clauses) in
  let p = maximize (posneg_count clauses) pvs in
  dpll (insert [p] clauses) or dpll (insert [negate p] clauses);;
```

```
let dpllsat fm = dpll(defcnfs fm);;

let dplltaut fm = not(dpllsat(Not fm));;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
dplltaut <<(p \/ (q /\  ~r)) /\ s>>;;
END_INTERACTIVE;;


(* ========================================================================= *)
(* Lógica de primer orden: sintaxis y semántica.                             *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Términos.                                                                 *)
(* ------------------------------------------------------------------------- *)

type term = Var of string
          | Fn of string * term list;;

(* ------------------------------------------------------------------------- *)
(* Fórmula de primer orden.                                                  *)
(* ------------------------------------------------------------------------- *)

type fol = R of string * term list;;

(* ------------------------------------------------------------------------- *)
(* Parsing de términos.                                                      *)
(* ------------------------------------------------------------------------- *)

let is_const_name s = forall numeric (explode s) || s = "nil";;

let rec parse_atomic_term vs inp =
  match inp with
    [] -> failwith "term expected"
  | "("::rest -> parse_bracketed (parse_term vs) ")" rest
  | "-"::rest -> papply (fun t -> Fn("-",[t])) (parse_atomic_term vs rest)
  | f::"("::")"::rest -> Fn(f,[]),rest
  | f::"("::rest ->
      papply (fun args -> Fn(f,args))
             (parse_bracketed (parse_list "," (parse_term vs)) ")" rest)
  | a::rest ->
```

```
      (if is_const_name a && not(mem a vs) then Fn(a,[]) else Var a),rest

and parse_term vs inp =
  parse_right_infix "::" (fun (e1,e2) -> Fn("::",[e1;e2]))
    (parse_right_infix "+" (fun (e1,e2) -> Fn("+",[e1;e2]))
       (parse_left_infix "-" (fun (e1,e2) -> Fn("-",[e1;e2]))
          (parse_right_infix "*" (fun (e1,e2) -> Fn("*",[e1;e2]))
             (parse_left_infix "/" (fun (e1,e2) -> Fn("/",[e1;e2]))
                (parse_left_infix "^" (fun (e1,e2) -> Fn("^",[e1;e2]))
                   (parse_atomic_term vs)))))) inp;;

let parset = make_parser (parse_term []);;

(* ------------------------------------------------------------------------- *)
(* Parsing de fórmulas.                                                      *)
(* ------------------------------------------------------------------------- *)

let parse_infix_atom vs inp =
  let tm,rest = parse_term vs inp in
  if exists (nextin rest) ["="; "<"; "<="; ">"; ">="] then
        papply (fun tm' -> Atom(R(hd rest,[tm;tm'])))
               (parse_term vs (tl rest))
  else failwith "";;

let parse_atom vs inp =
  try parse_infix_atom vs inp with Failure _ ->
  match inp with
  | p::"("::")"::rest -> Atom(R(p,[])),rest
  | p::"("::rest ->
      papply (fun args -> Atom(R(p,args)))
             (parse_bracketed (parse_list "," (parse_term vs)) ")" rest)
  | p::rest when p <> "(" -> Atom(R(p,[])),rest
  | _ -> failwith "parse_atom";;

let parse = make_parser
  (parse_formula (parse_infix_atom,parse_atom) []);;

let default_parser = parse;;

let secondary_parser = parset;;

(* ------------------------------------------------------------------------- *)
(* Printing de términos.                                                     *)
(* ------------------------------------------------------------------------- *)

let rec print_term prec fm =
```

```
  match fm with
    Var x -> print_string x
  | Fn("^",[tm1;tm2]) -> print_infix_term true prec 24 "^" tm1 tm2
  | Fn("/",[tm1;tm2]) -> print_infix_term true prec 22 " /" tm1 tm2
  | Fn("*",[tm1;tm2]) -> print_infix_term false prec 20 " *" tm1 tm2
  | Fn("-",[tm1;tm2]) -> print_infix_term true prec 18 " -" tm1 tm2
  | Fn("+",[tm1;tm2]) -> print_infix_term false prec 16 " +" tm1 tm2
  | Fn("::",[tm1;tm2]) -> print_infix_term false prec 14 "::" tm1 tm2
  | Fn(f,args) -> print_fargs f args

and print_fargs f args =
  print_string f;
  if args = [] then () else
   (print_string "(";
    open_box 0;
    print_term 0 (hd args); print_break 0 0;
    do_list (fun t -> print_string ","; print_break 0 0; print_term 0 t)
            (tl args);
    close_box();
    print_string ")")

and print_infix_term isleft oldprec newprec sym p q =
  if oldprec > newprec then (print_string "("; open_box 0) else ();
  print_term (if isleft then newprec else newprec+1) p;
  print_string sym;
  print_break (if String.sub sym 0 1 = " " then 1 else 0) 0;
  print_term (if isleft then newprec+1 else newprec) q;
  if oldprec > newprec then (close_box(); print_string ")") else ();;

let printert tm =
  open_box 0; print_string "<<|";
  open_box 0; print_term 0 tm; close_box();
  print_string "|>>"; close_box();;

#install_printer printert;;

(* ------------------------------------------------------------------------- *)
(* Printing de fórmulas.                                                     *)
(* ------------------------------------------------------------------------- *)

let print_atom prec (R(p,args)) =
  if mem p ["="; "<"; "<="; ">"; ">="] & length args = 2
  then print_infix_term false 12 12 (" "^p) (el 0 args) (el 1 args)
  else print_fargs p args;;

let print_fol_formula = print_qformula print_atom;;
```

```
#install_printer print_fol_formula;;

(* ------------------------------------------------------------------------- *)
(* Variables libres en términos y fórmulas.                                  *)
(* ------------------------------------------------------------------------- *)

let rec fvt tm =
  match tm with
    Var x -> [x]
  | Fn(f,args) -> unions (map fvt args);;

let rec var fm =
   match fm with
    False | True -> []
  | Atom(R(p,args)) -> unions (map fvt args)
  | Not(p) -> var p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> union (var p) (var q)
  | Forall(x,p) | Exists(x,p) -> insert x (var p);;

let rec fv fm =
  match fm with
    False | True -> []
  | Atom(R(p,args)) -> unions (map fvt args)
  | Not(p) -> fv p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> union (fv p) (fv q)
  | Forall(x,p) | Exists(x,p) -> subtract (fv p) [x];;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
<<(forall x. x < 2 ==> 2 * x <= 3) \/ false>>;;

<<forall x y. exists z. x < z /\ y < z>>;;

<< ~(forall x. P(x)) <=> exists y.  ~P(y)>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Semántica.                                                                *)
(* ------------------------------------------------------------------------- *)

let rec termval (domain,func,pred as m) v tm =
  match tm with
```

```
    Var(x) -> apply v x
  | Fn(f,args) -> func f (map (termval m v) args);;

let rec holds (domain,func,pred as m) v fm =
  match fm with
    False -> false
  | True -> true
  | Atom(R(r,args)) -> pred r (map (termval m v) args)
  | Not(p) -> not(holds m v p)
  | And(p,q) -> (holds m v p) & (holds m v q)
  | Or(p,q) -> (holds m v p) or (holds m v q)
  | Imp(p,q) -> not(holds m v p) or (holds m v q)
  | Iff(p,q) -> (holds m v p = holds m v q)
  | Forall(x,p) -> forall (fun a -> holds m ((x |-> a) v) p) domain
  | Exists(x,p) -> exists (fun a -> holds m ((x |-> a) v) p) domain;;

(* ------------------------------------------------------------------------- *)
(* Examples de interpretaciones particulares.                                *)
(* ------------------------------------------------------------------------- *)

let bool_interp =
  let func f args =
    match (f,args) with
      ("0",[]) -> false
    | ("1",[]) -> true
    | ("+",[x;y]) -> not(x = y)
    | ("*",[x;y]) -> x & y
    | _ -> failwith "uninterpreted function"
  and pred p args =
    match (p,args) with
      ("=",[x;y]) -> x = y
    | _ -> failwith "uninterpreted predicate" in
  ([false; true],func,pred);;

START_INTERACTIVE;;
holds bool_interp undefined <<forall x. (x = 0) \/ (x = 1)>>;;

let fm = <<forall x. ~(x = 0) ==> exists y. x * y = 1>>;;

holds bool_interp undefined fm;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Cierre universal de una fórmula.                                          *)
(* ------------------------------------------------------------------------- *)
```

```
let generalize fm = itlist mk_forall (fv fm) fm;;

(* ------------------------------------------------------------------------- *)
(* Sustitución en términos.                                                  *)
(* ------------------------------------------------------------------------- *)

let rec tsubst sfn tm =
  match tm with
    Var x -> tryapplyd sfn x tm
  | Fn(f,args) -> Fn(f,map (tsubst sfn) args);;

let rec variant x vars =
  if mem x vars then variant (x^"'") vars else x;;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
variant "x" ["y"; "z"];;

variant "x" ["x"; "y"];;

variant "x" ["x"; "x'"];;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Sustitución en fórmulas.                                                  *)
(* ------------------------------------------------------------------------- *)

let rec subst subfn fm =
  match fm with
    False -> False
  | True -> True
  | Atom(R(p,args)) -> Atom(R(p,map (tsubst subfn) args))
  | Not(p) -> Not(subst subfn p)
  | And(p,q) -> And(subst subfn p,subst subfn q)
  | Or(p,q) -> Or(subst subfn p,subst subfn q)
  | Imp(p,q) -> Imp(subst subfn p,subst subfn q)
  | Iff(p,q) -> Iff(subst subfn p,subst subfn q)
  | Forall(x,p) -> substq subfn mk_forall x p
  | Exists(x,p) -> substq subfn mk_exists x p

and substq subfn quant x p =
  let x' = if exists (fun y -> mem x (fvt(tryapplyd subfn y (Var y))))
                     (subtract (fv p) [x])
```

```
           then variant x (fv(subst (undefine x subfn) p)) else x in
  quant x' (subst ((x |-> Var x') subfn) p);;

(* ------------------------------------------------------------------------- *)
(* Examples.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
subst ("y" |=> Var "x") <<forall x. x = y>>;;

subst ("y" |=> Var "x") <<forall x x'. x = y ==> x = x'>>;;
END_INTERACTIVE;;


(* ========================================================================= *)
(* Formas prenexa y normales.                                                *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Simplificaciones.                                                         *)
(* ------------------------------------------------------------------------- *)

let simplify1 fm =
  match fm with
    Forall(x,p) -> if mem x (fv p) then fm else p
  | Exists(x,p) -> if mem x (fv p) then fm else p
  | _ -> psimplify1 fm;;

let rec simplify fm =
  match fm with
    Not p -> simplify1 (Not(simplify p))
  | And(p,q) -> simplify1 (And(simplify p,simplify q))
  | Or(p,q) -> simplify1 (Or(simplify p,simplify q))
  | Imp(p,q) -> simplify1 (Imp(simplify p,simplify q))
  | Iff(p,q) -> simplify1 (Iff(simplify p,simplify q))
  | Forall(x,p) -> simplify1(Forall(x,simplify p))
  | Exists(x,p) -> simplify1(Exists(x,simplify p))
  | _ -> fm;;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
simplify <<(forall x y. P(x) \/ (P(y) /\ false)) ==> exists z. Q>>;;
END_INTERACTIVE;;
```

```
(* ------------------------------------------------------------------------- *)
(* Forma normal negativa.                                                    *)
(* ------------------------------------------------------------------------- *)

let rec nnf fm =
  match fm with
    And(p,q) -> And(nnf p,nnf q)
  | Or(p,q) -> Or(nnf p,nnf q)
  | Imp(p,q) -> Or(nnf(Not p),nnf q)
  | Iff(p,q) -> Or(And(nnf p,nnf q),And(nnf(Not p),nnf(Not q)))
  | Not(Not p) -> nnf p
  | Not(And(p,q)) -> Or(nnf(Not p),nnf(Not q))
  | Not(Or(p,q)) -> And(nnf(Not p),nnf(Not q))
  | Not(Imp(p,q)) -> And(nnf p,nnf(Not q))
  | Not(Iff(p,q)) -> Or(And(nnf p,nnf(Not q)),And(nnf(Not p),nnf q))
  | Forall(x,p) -> Forall(x,nnf p)
  | Exists(x,p) -> Exists(x,nnf p)
  | Not(Forall(x,p)) -> Exists(x,nnf(Not p))
  | Not(Exists(x,p)) -> Forall(x,nnf(Not p))
  | _ -> fm;;

(* ------------------------------------------------------------------------- *)
(* Ejemplo.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
nnf <<(forall x. P(x))
      ==> ((exists y. Q(y)) <=> exists z. P(z) /\ Q(z))>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Forma normal prenexa.                                                     *)
(* ------------------------------------------------------------------------- *)

let rec pullquants fm =
  match fm with
    And(Forall(x,p),Forall(y,q)) ->
                          pullq(true,true) fm mk_forall mk_and x y p q
  | Or(Exists(x,p),Exists(y,q)) ->
                          pullq(true,true) fm mk_exists mk_or x y p q
  | And(Forall(x,p),q) -> pullq(true,false) fm mk_forall mk_and x x p q
  | And(p,Forall(y,q)) -> pullq(false,true) fm mk_forall mk_and y y p q
  | Or(Forall(x,p),q) ->  pullq(true,false) fm mk_forall mk_or x x p q
  | Or(p,Forall(y,q)) ->  pullq(false,true) fm mk_forall mk_or y y p q
  | And(Exists(x,p),q) -> pullq(true,false) fm mk_exists mk_and x x p q
```

```
  | And(p,Exists(y,q)) -> pullq(false,true) fm mk_exists mk_and y y p q
  | Or(Exists(x,p),q) ->  pullq(true,false) fm mk_exists mk_or x x p q
  | Or(p,Exists(y,q)) ->  pullq(false,true) fm mk_exists mk_or y y p q
  | _ -> fm

and pullq(l,r) fm quant op x y p q =
  let z = variant x (fv fm) in
  let p' = if l then subst (x |=> Var z) p else p
  and q' = if r then subst (y |=> Var z) q else q in
  quant z (pullquants(op p' q'));;

let rec prenex fm =
  match fm with
    Forall(x,p) -> Forall(x,prenex p)
  | Exists(x,p) -> Exists(x,prenex p)
  | And(p,q) -> pullquants(And(prenex p,prenex q))
  | Or(p,q) -> pullquants(Or(prenex p,prenex q))
  | _ -> fm;;

let pnf fm = prenex(nnf(simplify fm));;

(* ------------------------------------------------------------------------- *)
(* Example.                                                                  *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
nnf(simplify <<(forall x. P(x) \/ R(y)) ==> exists y z.
Q(y) \/  ~(exists z. P(z) /\ Q(z)) >>);;

pnf <<(forall x. P(x) \/ R(y))
      ==> exists y z. Q(y) \/ ~(exists z. P(z) /\ Q(z))>>;;
END_INTERACTIVE;;

(* ------------------------------------------------------------------------- *)
(* Obtener las funciones en un término o una fórmula.                        *)
(* ------------------------------------------------------------------------- *)

let rec funcs tm =
  match tm with
    Var x -> []
  | Fn(f,args) -> itlist (union ** funcs) args [f,length args];;

let functions fm =
  atom_union (fun (R(p,a)) -> itlist (union ** funcs) a []) fm;;

(* ------------------------------------------------------------------------- *)
```

```
(* Función de Skolem.                                                        *)
(* ------------------------------------------------------------------------- *)

let rec skolem fm fns =
  match fm with
    Exists(y,p) ->
        let xs = fv(fm) in
        let f = variant (if xs = [] then "c_"^y else "f_"^y) fns in
        let fx = Fn(f,map (fun x -> Var x) xs) in
        skolem (subst (y |=> fx) p) (f::fns)
  | Forall(x,p) -> let p',fns' = skolem p fns in Forall(x,p'),fns'
  | And(p,q) -> skolem2 (fun (p,q) -> And(p,q)) (p,q) fns
  | Or(p,q) -> skolem2 (fun (p,q) -> Or(p,q)) (p,q) fns
  | _ -> fm,fns

and skolem2 cons (p,q) fns =
  let p',fns' = skolem p fns in
  let q',fns'' = skolem q fns' in
  cons(p',q'),fns'';;

(* ------------------------------------------------------------------------- *)
(* Procedimiento completo.                                                   *)
(* ------------------------------------------------------------------------- *)

let askolemize fm =
  fst(skolem (nnf(simplify fm)) (map fst (functions fm)));;

let rec specialize fm =
  match fm with
    Forall(x,p) -> specialize p
  | _ -> fm;;

let skolemize fm = specialize(pnf(askolemize fm));;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
askolemize <<exists y. x < y ==> forall u. exists v. x * u < y * v>>;;

skolemize <<exists y. x < y ==> forall u. exists v. x * u < y * v>>;;

askolemize (pnf <<exists y. x < y ==> forall u. exists v. x * u < y * v>>);;
END_INTERACTIVE;;
```

```
(* ========================================================================= *)
(* Eliminación de cuantificadores.                                           *)
(* ========================================================================= *)


let qelim bfn x p =
  let cjs = conjuncts p in
  let ycjs,ncjs = partition (mem x ** fv) cjs in
  if ycjs = [] then p else
  let q = bfn (Exists(x,list_conj ycjs)) in
  itlist mk_and ncjs q;;

(* ------------------------------------------------------------------------- *)
(* Reducción del alcance de los cuantificadores.                             *)
(* ------------------------------------------------------------------------- *)

let separate x cjs =
  let yes,no = partition (mem x ** fv) cjs in
  if yes = [] then list_conj no
  else if no = [] then Exists(x,list_conj yes)
  else And(Exists(x,list_conj yes),list_conj no);;

let rec pushquant x p =
  if not (mem x (fv p)) then p else
  let djs = purednf(nnf p) in
  list_disj (map (separate x) djs);;

let rec miniscope fm =
  match fm with
    Not p -> Not(miniscope p)
  | And(p,q) -> And(miniscope p,miniscope q)
  | Or(p,q) -> Or(miniscope p,miniscope q)
  | Forall(x,p) -> Not(pushquant x (Not(miniscope p)))
  | Exists(x,p) -> pushquant x (miniscope p)
  | _ -> fm;;

(* ------------------------------------------------------------------------- *)
(* Función principal.                                                        *)
(* ------------------------------------------------------------------------- *)

let lift_qelim afn nfn qfn =
  let rec qelift vars fm =
    match fm with
    | Atom(R(_,_)) -> afn vars fm
    | Not(p) -> Not(qelift vars p)
```

```
    | And(p,q) -> And(qelift vars p,qelift vars q)
    | Or(p,q) -> Or(qelift vars p,qelift vars q)
    | Imp(p,q) -> Imp(qelift vars p,qelift vars q)
    | Iff(p,q) -> Iff(qelift vars p,qelift vars q)
    | Forall(x,p) -> Not(qelift vars (Exists(x,Not p)))
    | Exists(x,p) ->
          let djs = disjuncts(nfn(qelift (x::vars) p)) in
          list_disj(map (qelim (qfn vars) x) djs)
    | _ -> fm in
  fun fm -> simplify(qelift (fv fm) (miniscope fm));;

let cnnf lfn =
  let rec cnnf fm =
    match fm with
      And(p,q) -> And(cnnf p,cnnf q)
    | Or(p,q) -> Or(cnnf p,cnnf q)
    | Imp(p,q) -> Or(cnnf(Not p),cnnf q)
    | Iff(p,q) -> Or(And(cnnf p,cnnf q),And(cnnf(Not p),cnnf(Not q)))
    | Not(Not p) -> cnnf p
    | Not(And(p,q)) -> Or(cnnf(Not p),cnnf(Not q))
    | Not(Or(And(p,q),And(p',r))) when p' = negate p ->
        Or(cnnf (And(p,Not q)),cnnf (And(p',Not r)))
    | Not(Or(p,q)) -> And(cnnf(Not p),cnnf(Not q))
    | Not(Imp(p,q)) -> And(cnnf p,cnnf(Not q))
    | Not(Iff(p,q)) -> Or(And(cnnf p,cnnf(Not q)),
                          And(cnnf(Not p),cnnf q))
    | _ -> lfn fm in
  simplify ** cnnf ** simplify;;

(* ------------------------------------------------------------------------- *)
(* Igualdad.                                                                 *)
(* ------------------------------------------------------------------------- *)

let is_eq = function (Atom(R("=",_))) -> true | _ -> false;;

let dest_eq fm =
  match fm with
    Atom(R("=",[s;t])) -> s,t
  | _ -> failwith "dest_eq: not an equation";;

(* ------------------------------------------------------------------------- *)
(* Órdenes lineales densos.                                                  *)
(* ------------------------------------------------------------------------- *)

let afn_dlo vars fm =
  match fm with
```

```
    Atom(R("<=",[s;t])) -> Not(Atom(R("<",[t;s])))
  | Atom(R(">=",[s;t])) -> Not(Atom(R("<",[s;t])))
  | Atom(R(">",[s;t])) -> Atom(R("<",[t;s]))
  | _ -> fm;;

let lfn_dlo fm =
  match fm with
    Not(Atom(R("<",[s;t]))) -> Or(Atom(R("=",[s;t])),Atom(R("<",[t;s])))
  | Not(Atom(R("=",[s;t]))) -> Or(Atom(R("<",[s;t])),Atom(R("<",[t;s])))
  | _ -> fm;;

let dlobasic fm =
  match fm with
    Exists(x,p) ->
      let cjs = subtract (conjuncts p) [Atom(R("=",[Var x;Var x]))] in
      try let eqn = find is_eq cjs in
          let s,t = dest_eq eqn in
          let y = if s = Var x then t else s in
          list_conj(map (subst (x |=> y)) (subtract cjs [eqn]))
      with Failure _ ->
          if mem (Atom(R("<",[Var x;Var x]))) cjs then False else
          let lefts,rights =
            partition (fun (Atom(R("<",[s;t]))) -> t = Var x) cjs in
          let ls = map (fun (Atom(R("<",[l;_]))) -> l) lefts
          and rs = map (fun (Atom(R("<",[_;r]))) -> r) rights in
          list_conj(allpairs (fun l r -> Atom(R("<",[l;r]))) ls rs)
  | _ -> failwith "dlobasic";;

let quelim_dlo =
  lift_qelim afn_dlo (dnf ** cnnf lfn_dlo) (fun v -> dlobasic);;

(* ------------------------------------------------------------------------- *)
(* Ejemplos.                                                                 *)
(* ------------------------------------------------------------------------- *)

START_INTERACTIVE;;
quelim_dlo <<exists z. z < x /\ z < y>>;;

quelim_dlo <<exists z. x < z /\ z < y>>;;

quelim_dlo <<(forall x. x < a ==> x < b)>>;;

quelim_dlo <<forall a b. (forall x. x < a ==> x < b) <=> a <= b>>;;
END_INTERACTIVE;;
```