Logica Proposicional en Isabelle/HOL

Sofía Santiago Fernández

actualizado el 18 de octubre de 2019

Resumen

Añadir introducción

Índice

1 Sintaxis

notation *insert* (- ▷ - [*56*,*55*] *55*)

En este apartado vamos a desarrollar los elementos de la sintaxis junto con varios resultados sobre los mismos. La lógica proposicional cuenta, fundamentalmente con:

Alfabeto: Se trata de una lista infinita de variables proposicionales, consideradas como expresiones cuya estructura interna no vamos a analizar.

Conectivas: Conectores que interactúan con los elementos del alfabeto. Pueden ser monarias que afectan a un único elemento o binarias que afectan a dos. En el primer grupo se encuentra le negación ($\neg$) y en el segundo vamos a considerar la conjunción ($\wedge$), la disyunción ($\vee$) y la implicación ($\longrightarrow$).

De este modo, informalmente diremos que una fórmula es el resultado de relacionar los elementos del alfabeto mediante las conectivas definidas anteriormente. En primer lugar, podemos definir las fórmulas atómicas como el tipo de fórmulas más básico, formadas únicamente por una variable porposicional del alfabeto. Por abuso de notación llamaremos *átomos* a las variables proposicionales del alfabeto. Aunque son intuitivamente equivalentes a las fórmulas atómicas, debemos recalcar su diferencia, pues los átomos son los elementos del alfabeto y las fórmulas atómicas son construcciones básicas de ellos. Este apunte es fundamental para entender el tipo correspondiente de fórmulas en nuestro programa.

En Isabelle, las fórmulas son entendidas como un árbol con distintos tipos de nodos, que son las conectivas, y hojas, que serán las fórmulas atómicas. De este modo, se define el tipo de las fórmulas como sigue.

datatype (*atoms*: *'a*) *formula* =
  Atom 'a
| *Bot* (⊥)
| *Not 'a formula* (¬)
| *And 'a formula 'a formula* (**infix** ∧ *68*)
| *Or 'a formula 'a formula* (**infix** ∨ *68*)
| *Imp 'a formula 'a formula* (**infixr** → *68*)

Como podemos observar en la definición, *formula* es un tipo de datos recursivo que se entiende como un árbol que relaciona elementos de un tipo *'a* cualquiera del alfabeto proposicional. En ella aparecen los siguientes elementos:

Constructores :

1. *Atom* :: *'a formula*
2. *Not* :: *'a* ⇒ *'a formula* ⇒ *'a formula*
3. *And* :: *'a* ⇒ *'a formula* ⇒ *'a formula* ⇒ *'a formula*
4. *Or* :: *'a* ⇒ *'a formula* ⇒ *'a formula* ⇒ *'a formula*
5. *Imp* :: *'a* ⇒ *'a formula* ⇒ *'a formula* ⇒ *'a formula*
6. *Bot* :: *'a formula*

Función de conjunto : *atoms* :: *'a formula* ⇒ *'a set*

Podemos observar que se define simultáneamente la función *atoms* de modo que al aplicarla sobre una fórmula nos devuelve el conjunto de los átomos que la componen. En particular, *Atom 'a* es la construcción de las fórmulas atómicas. *Bot* es un constructor que para cada tipo *'a* cualquiera construye la fórmula falsa cuyo símbolo queda retratado en la definición.

Observemos que para emplear *Bot* en solitario es necesario especificar el tipo $'a$.

value(*Bot*::*nat formula*)

Veamos ejemplos de obtención del conjunto de las variables proposicionales de las fórmulas atómicas y de negación.

value *atoms* (*Atom p*) = {*p*}

value *atoms* (*Not* (*Atom p*)) = {*p*}

En particular, al aplicar *atoms* sobre la construcción *Bot* nos devuelve el conjunto vacío, pues como hemos señalado, no contiene ninguna variable del alfabeto.

lemma *atoms Bot* = {}
by *auto*

lemma *atoms* (*Or* (*Atom p*) *Bot*) = {*p*}
by *auto*

El resto de elementos que aparecen son equivalentes a las conectivas binarias y la negación. Cabe señalar que el término *infix* que precede al símbolo de notación de los nodos nos señala que son infijos, y *infixr* se trata de un infijo asociado a la derecha. A continuación vamos a incluir el ejemplo de fórmula: $(p \longrightarrow q) \vee r$ y su árbol de formación correspondiente.

value *Or* (*Imp* (*Atom p*) (*Atom q*)) (*Atom r*)

(Aquí debería salir el árbol pero no sé hacerlo)

Por otro lado, veamos cómo actúa la función *atoms* sobre fórmulas construidas con conectivas binarias, señalando los casos en que interactúan variables distintas y repetidas. Como se observa, por definición de conjunto, no contiene elementos repetidos.

lemma *atoms* (*Or* (*Imp* (*Atom p*) (*Atom q*)) (*Atom r*)) = {*p*,*q*,*r*}
by *auto*

lemma *atoms* (*Or* (*Imp* (*Atom r*) (*Atom p*)) (*Atom r*)) = {*p*,*r*}
by *auto*

En esta sección, para demostrar los distintos resultados utilizaremos la táctica *induction*, que hace uso del esquema de indución. Para el tipo de las fórmulas, el esquema inductivo se aplicará en cada uno de los casos de los constructores, desglosándose así seis casos correspondientes a las distintas conectivas, fórmula atómica y *Bot*. Además, todas las demostraciones sobre casos de conectivas binarias son equivalentes en esta sección, pues la construcción sintáctica de fórmulas es idéntica entre ellas. Estas se diferencian esencialmente por la semántica, que veremos en la siguiente sección. Por tanto, para simplificar las demostraciones sintácticas detalladas más extensas, daré un nuevo tipo equivalente: *formula-simp*.

datatype (*atoms-s*: *'a*) *formula-simp* =
  Atom-s 'a
 | *Bot-s* (*Falso*)
 | *Mon 'a formula-simp* (*Neg*)
 | *Bi 'a formula-simp 'a formula-simp* (**infix** $*$ *68*)

De este modo, se consideran todas las conectivas binarias dentro de un mismo caso de constructor *Bi* con notación $*$, y la conectiva mónica como *Mon*, de notación *Neg*. Para que no haya confusión, he renombrado la notación del equivalente de *Bot* como *Falso*. De este modo, en la inducción sobre esta nueva definición se deglosarán únicamente cuatro casos.

Análogamente, a lo largo de la sección definiré si es necesario la versión simplificada de otros tipos que se incluyan. La demostración automática aparecerá enunciada y demostrada con la definición original de *formula*.

Llegamos así al primer resultado de este apartado:

Lema 1.1 *Los átomos de cualquier fórmula conforman un conjunto finito.*

En Isabelle, se traduce del siguiente modo.

lemma *atoms-finite*[*simp*,*intro*!]: *finite* (*atoms F*)
 oops

El lema anterior contiene la notación *simp*,*intro*! a continuación del título para incluir este resultado en las tácticas automática (mediante *intro*!) y de simplificación (mediante *simp*). Esto ocurrirá en varios resultados de esta sección.

Por otro lado, el enunciado contiene la defición *finite S*, perteneciente a la teoría FiniteSet.thy. Se trata de una definición inductiva que nos permite la demostración del lema por simplificacion ya que dentro de ella, las reglas que especifica se han añadido como tácticas de *simp* e *intro*!.

inductive finite :: $'a$ *set* $\Rightarrow$ *bool*
where
emptyI [*simp*, *intro*!]: *finite* {}
| *insertI* [*simp*, *intro*!]: *finite A* $\Longrightarrow$ *finite* (*insert a A*)

Veamos la prueba detallada del resultado que, aunque se resulve fácilmente por simplificación, nos muestra un ejemplo claro de la estructura inductiva que nos acompañará en las siguientes demostraciones de este apartado.

lemma *atoms-finite-detallada*: *finite* (*atoms F*)
proof (*induction F*)
case (*Atom x*)
then show *?case* **by** *simp*
next
case *Bot*
  then show *?case* **by** *simp*
next
  case (*Not F*)
  then show *?case* **by** *simp*
next
  case (*And F1 F2*)
  then show *?case* **by** *simp*
next
  case (*Or F1 F2*)
  then show *?case* **by** *simp*
next
  case (*Imp F1 F2*)
  then show *?case* **by** *simp*
qed

Las demostraciones aplicativa y automática son las siguientes respectivamente.

lemma *atoms-finite-aplicativa*: *finite* (*atoms F*)
  apply (*induction F*)
  apply *simp-all*
 done

lemma *atoms-finite*[*simp*,*intro*!]: *finite* (*atoms F*)
  by (*induction F*; *simp*)

1.1 Subfórmulas

Otra construcción natural a partir de la definición de fórmulas son las subfórmulas.

Definición 1.2 *La lista de las subfórmulas de una fórmula F ($Subf(F)$) se define recursivamente por:*

1. *$[F]$ si F es atómica.*
2. *$[Bot]$ si F es Bot.*
3. *$F\#Subf(G)$ si F es $\neg G$.*
4. *$F\#Subf(G)@Subf(H)$ si F es $G{*}H$ donde $*$ es cualquier conectiva binaria.*

En la definición anterior, # es el operador que añade un elemento al comienzo de una lista y @ concatena varias listas. Análogamente, vamos a definir la función primitiva recursiva *subformulae*, que nos dará una lista de todas las subfórmulas de una fórmula original obtenidas recursivamente.

primrec *subformulae* :: *'a formula* ⇒ *'a formula list* **where**
subformulae $\bot = [\bot]$ |
subformulae (*Atom k*) = [*Atom k*] |
subformulae (*Not F*) = *Not F* # *subformulae F* |
subformulae (*And F G*) = *And F G* # *subformulae F* @ *subformulae G* |
subformulae (*Imp F G*) = *Imp F G* # *subformulae F* @ *subformulae G* |
subformulae (*Or F G*) = *Or F G* # *subformulae F* @ *subformulae G*

Su definición simplificada equivalente es la siguiente.

primrec *subformulae-s* :: *'a formula-simp* ⇒ *'a formula-simp list* **where**
subformulae-s (*Bot-s*) = [*Bot-s*] |
subformulae-s (*Atom-s k*) = [*Atom-s k*] |
subformulae-s (*Mon F*) = *Mon F* # *subformulae-s F* |
subformulae-s (*Bi F G*) = *Bi F G* # *subformulae-s F* @ *subformulae-s G*

Siguiendo con los ejemplos, observemos cómo actúa *subformulae* en las distintas fórmulas. En primer lugar, veamos los casos base de fórmulas atómicas y con conectiva de negación.

value *subformulae* (*Atom p*) = [*Atom p*]

value *subformulae* (*Not* (*Atom p*)) = [*Not* (*Atom p*), *Atom p*]

A continuación, una fórmula con conectivas binarias y variables todas distintas.

value *subformulae* (*Or* (*Imp* (*Atom p*) (*Atom q*)) (*Atom r*)) =
[(*Atom p* $\longrightarrow$ *Atom q*) $\vee$ *Atom r*, *Atom p* $\longrightarrow$ *Atom q*, *Atom p*, *Atom q*, *Atom r*]

En particular, al tratarse de una lista pueden aparecer elementos repetidos como se muestra a continuación.

value *subformulae* (*Or* (*Atom p*) (*Atom p*)) =
[*Or* (*Atom p*) (*Atom p*), *Atom p*, *Atom p*]

value *subformulae* (*Or* (*Atom p*) (*Atom p*)) =
[*Or* (*Atom p*) (*Atom p*), *Atom p*] = *False*

Veamos su valor en presencia de *Bot*.

value *subformulae* (*And* (*Atom p*) *Bot*) =
[*And* (*Atom p*) *Bot*, *Atom p*, *Bot*]

A partir de esta definición, aparecen varios resultados sencillos que demostraremos siguiendo tácticas similares a las empleadas en el lema anterior. Como se ha argumentado anteriormente, para resumir las demostraciones detalladas se harán mediante las definiciones simplificadas de los tipos. Además, trabajaremos con conjuntos en vez de listas, pues poseen ventajas como la eliminación de elementos repetidos o las operaciones de conjuntos. De este modo, definimos *setSubformulae*, que convierte en un tipo conjunto la lista de subfórmulas anterior. Añadimos también su versión simplificada.

abbreviation *setSubformulae* :: *'a formula* $\Rightarrow$ *'a formula set* **where**
setSubformulae F $\equiv$ *set* (*subformulae F*)

abbreviation *setSubformulae-s* :: *'a formula-simp* $\Rightarrow$ *'a formula-simp set* **where**
setSubformulae-s F $\equiv$ *set* (*subformulae-s F*)

De este modo, observemos la diferencia de repetición con el ejemplo anterior.

value *setSubformulae* (*Or* (*Atom p*) (*Atom p*)) = {*Or* (*Atom p*) (*Atom p*), *Atom p*}

lemma *setSubformulae* $(Or\ (Imp\ (Atom\ p)\ (Atom\ q))\ (Atom\ r)) =$
$\{(Atom\ p \rightarrow Atom\ q) \vee Atom\ r,\ Atom\ p \rightarrow Atom\ q,\ Atom\ p,\ Atom\ q,\ Atom\ r\}$
by *auto*

Como hemos señalado, utilizaremos varios resultados de la teoría de conjuntos definida en Isabelle como Set.thy. Voy a especificar el esquema de las usadas en esta sección que no están incluidas en las tácticas de simplificación para aclarar las demostraciones detalladas que presentaré en este apartado.

$$\frac{A \subseteq B \wedge B \subseteq C}{A \subseteq C} \qquad (\textit{subset-trans})$$

$$\frac{c \in A \wedge A \subseteq B}{c \in B} \qquad (\textit{rev-subsetD})$$

Además, definiré alguna propiedad más que no aparece en la teoría de Isabelle y que utilizaremos con frecuencia. Su demostración será la automática.

lemma *subContUnion1*: $A = B \cup C \Longrightarrow B \subseteq A$
by *auto*

lemma *subContUnion2*: $A = B \cup C \Longrightarrow C \subseteq A$
by *auto*

lemma *subContUnionRev1*: $A \cup B \subseteq C \Longrightarrow A \subseteq C$
by *auto*

lemma *subContUnionRev2*: $A \cup B \subseteq C \Longrightarrow B \subseteq C$
by *auto*

lemma *subConts*: $[\![A \subseteq B;\ C \subseteq D]\!] \Longrightarrow A \cup C \subseteq B \cup D$
by *auto*

Una vez aclarada la nueva función de conjuntos, vamos a demostrar el siguiente lema sirviéndonos de ella.

Lema 1.3 *El conjunto de los átomos de una fórmula está contenido en el conjunto de subfórmulas de la misma, es decir, las fórmulas atómicas son subfórmulas.*

En Isabelle, se especifica como sigue.

lemma *atoms-are-subformulae*: *Atom* ‘ *atoms* *F* $\subseteq$ *setSubformulae* *F*
  oops

Este resultado es especialmente interesante para aclarar la naturaleza de la función *atoms* aplicada a una fórmula. De este modo, *Atom* ‘ *atoms* *F* se encarga de construir las fórmulas atómicas a partir de cada uno de los elementos del conjunto de átomos de la fórmula *F*, creando un conjunto de fórmulas atómicas. Para ello emplea el infijo ‘ definido como notación abreviada de (‘) que calcula la imagen de un conjunto en la teoría Set.thy.

$$f \text{ ‘ } A = \{y \mid \exists x{\in}A.\ y = f\ x\} \qquad (\textit{image-def})$$

Para aclarar su funcionamiento, veamos ejemplos para distintos casos de fórmulas.

value *Atom* ‘*atoms* (*Or* (*Atom* *p*) *Bot*) = {*Atom* *p*}

lemma *Atom* ‘*atoms* (*Or* (*Imp* (*Atom* *p*) (*Atom* *q*)) (*Atom* *r*)) = {*Atom* *p*, *Atom* *q*, *Atom* *r*}
  by *auto*

lemma *Atom* ‘*atoms* (*Or* (*Imp* (*Atom* *p*) (*Atom* *p*)) (*Atom* *r*)) = {*Atom* *p*, *Atom* *r*}
  by *auto*

Además, esta función tiene la siguiente propiedad sobre la unión de conjuntos que utilizaré en la demostración.

$$f \text{ ‘ } (A \cup B) = f \text{ ‘ } A \cup f \text{ ‘ } B \qquad (\textit{image-Un})$$

Una vez hecha la aclaración anterior, comencemos la demostración detallada simplificada, que seguirá el esquema inductivo señalado con anterioridad.

lemma *atoms-are-subformulae-detallada-s*: *Atom-s* ‘ *atoms-s* *F* $\subseteq$ *setSubformulae-s* *F*
proof (*induction* *F*)
case (*Atom-s* *x*)
  then show *?case* **by** *simp*
next

case *Bot-s*
then show *?case* **by** *simp*
next
case (*Mon F*)
assume *H*:*Atom-s* ‘ *atoms-s F* $\subseteq$ *setSubformulae-s F*
show *Atom-s* ‘ *atoms-s* (*Mon F*) $\subseteq$ *setSubformulae-s* (*Mon F*)
proof −
have *setSubformulae-s* (*Mon F*) = {*Mon F*} $\cup$ *setSubformulae-s F* **by** *simp*
then have *1*:*setSubformulae-s F* $\subseteq$ *setSubformulae-s* (*Mon F*) **by** (*rule subContUnion2*)
also have *Atom-s* ‘ *atoms-s F* = *Atom-s* ‘ *atoms-s* (*Mon F*) **by** *simp*
then have *Atom-s* ‘ *atoms-s* (*Mon F*) $\subseteq$ *setSubformulae-s F* **using** *H* **by** *simp*
then show *Atom-s* ‘ *atoms-s* (*Mon F*) $\subseteq$ *setSubformulae-s* (*Mon F*) **using** *1* **by** (*rule subset-trans*)
qed
next
case (*Bi F1 F2*)
assume *H1*:*Atom-s* ‘ *atoms-s F1* $\subseteq$ *setSubformulae-s F1*
assume *H2*:*Atom-s* ‘ *atoms-s F2* $\subseteq$ *setSubformulae-s F2*
show *Atom-s* ‘ *atoms-s* (*Bi F1 F2*) $\subseteq$ *setSubformulae-s* (*Bi F1 F2*)
proof −
have *2*:(*Atom-s* ‘ *atoms-s F1*) $\cup$ (*Atom-s* ‘ *atoms-s F2*) $\subseteq$ *setSubformulae-s F1* $\cup$ *setSubformulae-s F2*
using *H1 H2* **by** (*rule subConts*)
have *setSubformulae-s* (*Bi F1 F2*) = {*Bi F1 F2*} $\cup$ (*setSubformulae-s F1* $\cup$ *setSubformulae-s F2*) **by** *simp*
then have *3*:*setSubformulae-s F1* $\cup$ *setSubformulae-s F2* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **by** (*rule subContUnion2*)
then have *setSubformulae-s F1* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **by** *simp*
have *setSubformulae-s F2* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *3* **by** *simp*
also have *atoms-s* (*Bi F1 F2*) = *atoms-s F1* $\cup$ *atoms-s F2* **by** *simp*
then have *Atom-s* ‘ *atoms-s* (*Bi F1 F2*) = *Atom-s* ‘ (*atoms-s F1* $\cup$ *atoms-s F2*) **by** *simp*
also have ... = *Atom-s* ‘ *atoms-s F1* $\cup$ *Atom-s* ‘ *atoms-s F2* **by** (*rule image-Un*)
then have *Atom-s* ‘ *atoms-s* (*Bi F1 F2*) = *Atom-s* ‘ *atoms-s F1* $\cup$ *Atom-s* ‘ *atoms-s F2* **by** *simp*
then have *Atom-s* ‘ *atoms-s* (*Bi F1 F2*) $\subseteq$ *setSubformulae-s F1* $\cup$

setSubformulae-s F2 **using** *2* **by** *simp*
  then show *Atom-s ‘ atoms-s* (*Bi F1 F2*) $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *3* **by** (*rule subset-trans*)
 qed
qed

Mostremos también la demostración automática con la definición original.

lemma *atoms-are-subformulae*: *Atom ‘ atoms F* $\subseteq$ *setSubformulae F*
 by (*induction F*) *auto*

Otro resultado de esta sección es la propia pertenencia de una fórmula en el conjunto de sus subfórmulas.

Lema 1.4 *La propia fórmula pertence a la lista de sus subfórmulas, es decir:* $F \in Subf(F)$*.*

A continuación incluimos el enunciado del lema con su demostración automática. Las demostraciones detallada y aplicativa son análogas a las del primer lema de la sección, utilizando únicamente inducción y simplificación. Para facilitar pruebas posteriores, vamos a añadir la regla a las tácticas de simplificación.

lemma *subformulae-self*[*simp*,*intro*]: *F* $\in$ *setSubformulae F*
 by (*induction F*) *simp-all*

lemma *subformulae-self-s*[*simp*,*intro*]: *F* $\in$ *setSubformulae-s F*
 by (*induction F*) *simp-all*

La siguiente propiedad declara que el conjunto de átomos de una subfórmula está contenido en el conjunto de átomos de la propia fórmula.

Lema 1.5 *Sea* $G \in Subf(F)$*, entonces el conjunto de los átomos de G está contenido en los de F.*

Traducido a Isabelle:

lemma *subformula-atoms*: *G* $\in$ *setSubformulae F* $\Longrightarrow$ *atoms G* $\subseteq$ *atoms F*
 oops

Veamos su demostración estructurada con la definición simplificada para resumir los casos de conectivas binarias.

lemma *subformula-atoms-estructurada-s*: $G \in$ *setSubformulae-s F* $\Longrightarrow$ *atoms-s G* $\subseteq$ *atoms-s F*
proof (*induction F*)
  case (*Atom-s x*)
  then show *?case* **by** *simp*
next
  case *Bot-s*
  then show *?case* **by** *simp*
next
  case (*Mon F*)
  assume *H1*: $G \in$ *setSubformulae-s* (*Mon F*)
  assume *H2*: $G \in$ *setSubformulae-s F* $\Longrightarrow$ *atoms-s G* $\subseteq$ *atoms-s F*
  show *atoms-s G* $\subseteq$ *atoms-s* (*Mon F*)
  proof (*cases G = Mon F*)
    case *True*
    then have *G = Mon F* **by** *simp*
    then show *atoms-s G* $\subseteq$ *atoms-s* (*Mon F*) **by** *simp*
  next
    case *False*
    then have *1*:*G* $\neq$ *Mon F* **by** *simp*
    have *setSubformulae-s* (*Mon F*) = {*Mon F*} $\cup$ *setSubformulae-s F* **by** *simp*
    then have *2*:*G* $\in$ *setSubformulae-s F* **using** *1 H1* **by** *simp*
    have *atoms-s F = atoms-s* (*Mon F*) **by** *simp*
    then show *atoms-s G* $\subseteq$ *atoms-s* (*Mon F*) **using** *2 H2* **by** *simp*
  qed
next
  case (*Bi F1 F2*)
  assume *H3*: $G \in$ *setSubformulae-s* (*Bi F1 F2*)
  assume *H4*: $G \in$ *setSubformulae-s F1* $\Longrightarrow$ *atoms-s G* $\subseteq$ *atoms-s F1*
  assume *H5*: $G \in$ *setSubformulae-s F2* $\Longrightarrow$ *atoms-s G* $\subseteq$ *atoms-s F2*
  then show *atoms-s G* $\subseteq$ *atoms-s* (*Bi F1 F2*)
  proof (*cases G = Bi F1 F2*)
    case *True*
    then have *G = Bi F1 F2* **by** *simp*
    then show *atoms-s G* $\subseteq$ *atoms-s* (*Bi F1 F2*) **by** *simp*
  next
    case *False*
    then have *3*:*G* $\neq$ *Bi F1 F2* **by** *simp*
    have *setSubformulae-s* (*Bi F1 F2*) = {*Bi F1 F2*} $\cup$ *setSubformulae-s F1* $\cup$ *setSubformulae-s F2* **by** *simp*

then have *4*:$G \in$ *setSubformulae-s F1* $\cup$ *setSubformulae-s F2* **using** *3 H3* **by** *simp*
have *5*:*atoms-s* (*Bi F1 F2*) = *atoms-s F1* $\cup$ *atoms-s F2* **by** *simp*
then show *atoms-s G* $\subseteq$ *atoms-s* (*Bi F1 F2*)
proof (*cases G* $\in$ *setSubformulae-s F1*)
case *True*
then have $G \in$ *setSubformulae-s F1* **by** *simp*
then have *6*:*atoms-s G* $\subseteq$ *atoms-s F1* **using** *H4* **by** *simp*
have *7*:*atoms-s F1* $\subseteq$ *atoms-s* (*Bi F1 F2*) **using** *5* **by** (*rule subContUnion1*)
show *atoms-s G* $\subseteq$ *atoms-s* (*Bi F1 F2*) **using** *6 7* **by** (*rule subset-trans*)
next
case *False*
then have $G \notin$ *setSubformulae-s F1* **by** *simp*
then have $G \in$ *setSubformulae-s F2* **using** *4* **by** *simp*
then have *8*:*atoms-s G* $\subseteq$ *atoms-s F2* **using** *H5* **by** *simp*
have *9*:*atoms-s F2* $\subseteq$ *atoms-s* (*Bi F1 F2*) **using** *5* **by** *simp*
show *atoms-s G* $\subseteq$ *atoms-s* (*Bi F1 F2*) **using** *8 9* **by** (*rule subset-trans*)
qed
qed
qed

Por último, su demostración aplicativa y automática con la definición no simplificada.

lemma *subformula-atoms-aplicativa*: $G \in$ *setSubformulae F* $\Longrightarrow$ *atoms G* $\subseteq$ *atoms F*
apply (*induction F*)
apply *auto*
done

lemma *subformula-atoms*: $G \in$ *setSubformulae F* $\Longrightarrow$ *atoms G* $\subseteq$ *atoms F*
by (*induction F*) *auto*

A continuación voy a introducir un lema que no pertenece a la teoría original de Isabelle pero facilita las siguientes demostraciones detalladas mediante contenciones en cadena.

Lema 1.6 *Sea* $G \in SubfSet(F)$ *entonces* $SubfSet(G) \subseteq SubSet(F)$.

Para que la propiedad de contención esté bien definida, considero $SubfSet(\cdot)$ el conjunto equivalente a *setSubformulae* aplicado a una fórmula. Veamos las demostraciones estructurada simplificada y automática del mismo.

lemma *subsubformulae-estructurada-s*: $G \in$ *setSubformulae-s F* $\Longrightarrow$ *setSubformulae-s G* $\subseteq$ *setSubformulae-s F*
proof (*induction F*)
case (*Atom-s x*)
then show *?case* **by** *simp*
next
case *Bot-s*
then show *?case* **by** *simp*
next
case (*Mon F*)
assume *H1*: $G \in$ *setSubformulae-s F* $\Longrightarrow$ *setSubformulae-s G* $\subseteq$ *setSubformulae-s F*
assume *H2*: $G \in$ *setSubformulae-s* (*Mon F*)
then show *setSubformulae-s G* $\subseteq$ *setSubformulae-s* (*Mon F*)
proof (*cases G = Mon F*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $G \neq$ *Mon F* **by** *simp*
then have $G \in$ *setSubformulae-s F* **using** *H2* **by** *simp*
then have *1*:*setSubformulae-s G* $\subseteq$ *setSubformulae-s F* **using** *H1* **by** *simp*
have *setSubformulae-s* (*Mon F*) = {*Mon F*} $\cup$ *setSubformulae-s F* **by** *simp*
then have *2*:*setSubformulae-s F* $\subseteq$ *setSubformulae-s* (*Mon F*) **by** (*rule subContUnion2*)
show *setSubformulae-s G* $\subseteq$ *setSubformulae-s* (*Mon F*) **using** *1 2* **by** (*rule subset-trans*)
qed
next
case (*Bi F1 F2*)
assume *H3*: $G \in$ *setSubformulae-s F1* $\Longrightarrow$ *setSubformulae-s G* $\subseteq$ *setSubformulae-s F1*
assume *H4*: $G \in$ *setSubformulae-s F2* $\Longrightarrow$ *setSubformulae-s G* $\subseteq$ *setSubformulae-s F2*
assume *H5*: $G \in$ *setSubformulae-s* (*Bi F1 F2*)
have *4*:*setSubformulae-s* (*Bi F1 F2*) = {*Bi F1 F2*} $\cup$ (*setSubformulae-s F1* $\cup$ *setSubformulae-s F2*) **by** *simp*
then show *setSubformulae-s G* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*)
proof (*cases G = Bi F1 F2*)

case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *5*:$G \neq$ *Bi F1 F2* **by** *simp*
have *6*:*setSubformulae-s F1* $\cup$ *setSubformulae-s F2* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *4* **by** (*rule subContUnion2*)
then show *setSubformulae-s G* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*)
proof (*cases G* $\in$ *setSubformulae-s F1*)
case *True*
then have *G* $\in$ *setSubformulae-s F1* **by** *simp*
then have *7*:*setSubformulae-s G* $\subseteq$ *setSubformulae-s F1* **using** *H3* **by** *simp*
have *8*:*setSubformulae-s F1* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *6* **by** (*rule subContUnionRev1*)
show *setSubformulae-s G* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *7 8* **by** (*rule subset-trans*)
next
case *False*
then have *9*:*G* $\notin$ *setSubformulae-s F1* **by** *simp*
have *G* $\in$ *setSubformulae-s F1* $\cup$ *setSubformulae-s F2* **using** *5 H5* **by** *simp*
then have *G* $\in$ *setSubformulae-s F2* **using** *9* **by** *simp*
then have *10*:*setSubformulae-s G* $\subseteq$ *setSubformulae-s F2* **using** *H4* **by** *simp*
have *11*:*setSubformulae-s F2* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *6* **by** *simp*
show *setSubformulae-s G* $\subseteq$ *setSubformulae-s* (*Bi F1 F2*) **using** *10 11* **by** (*rule subset-trans*)
qed
qed
qed

lemma *subformulae-setSubformulae*:*G* $\in$ *setSubformulae F* $\Longrightarrow$ *setSubformulae G* $\subseteq$ *setSubformulae F*
by (*induction F*) *auto*

El siguiente lema nos da la noción de transitividad de contención en cadena de las subfórmulas, de modo que la subfórmula de una subfórmula es del mismo modo subfórmula de la mayor. Es un resultado sencillo derivado de la estructura de árbol de formación que siguen las fórmulas según las hemos definido.

Lema 1.7 *Sea* $G \in Subf(F)$ *y* $H \in Subf(G)$*, entonces* $H \in Subf(F)$*.*

La demostración estructurada se hace de manera sencilla con el resultado introducido anteriormente. Veamos su prueba según las distintas tácticas.

lemma *subsubformulae-estruct*:
 assumes $G \in$ *setSubformulae F*
  $H \in$ *setSubformulae G*
 shows $H \in$ *setSubformulae F*
proof −
 have *1*:*setSubformulae G* ⊆ *setSubformulae F* **using** *assms*(*1*) **by** (*rule subformulae-setSubformulae*)
 have *setSubformulae H* ⊆ *setSubformulae G* **using** *assms*(*2*) **by** (*rule subformulae-setSubformulae*)
 then have *2*:*setSubformulae H* ⊆ *setSubformulae F* **using** *1* **by** (*rule subset-trans*)
 have $H \in$ *setSubformulae H* **by** *simp*
 then show $H \in$ *setSubformulae F* **using** *2* **by** (*rule rev-subsetD*)
qed

lemma *subsubformulae-aplic*: $G \in$ *setSubformulae F* $\Longrightarrow$ $H \in$ *setSubformulae G* $\Longrightarrow$ $H \in$ *setSubformulae F*
 oops

lemma *subsubformulae*: $G \in$ *setSubformulae F* $\Longrightarrow$ $H \in$ *setSubformulae G* $\Longrightarrow$ $H \in$ *setSubformulae F*
 by (*induction F*; *force*)

A continuación, la versión del lema con definiciones simplificadas, pues será utilizada para la siguiente prueba.

lemma *subsubformulae-s*: $G \in$ *setSubformulae-s F* $\Longrightarrow$ $H \in$ *setSubformulae-s G* $\Longrightarrow$ $H \in$ *setSubformulae-s F*
 by (*induction F*; *force*)

A continuación presentamos otro resultado que relaciona las subfórmulas de una fórmula según las conectivas con operaciones sobre los conjuntos de subfórmulas de cada parte.

lemma *subformulas-in-subformulas*:
$G \wedge H \in$ *setSubformulae* $F \Longrightarrow G \in$ *setSubformulae* $F \wedge H \in$ *setSubformulae* F
$G \vee H \in$ *setSubformulae* $F \Longrightarrow G \in$ *setSubformulae* $F \wedge H \in$ *setSubformulae* F
$G \rightarrow H \in$ *setSubformulae* $F \Longrightarrow G \in$ *setSubformulae* $F \wedge H \in$ *setSubformulae* F
$\neg\ G \in$ *setSubformulae* $F \Longrightarrow G \in$ *setSubformulae* F
oops

Como podemos observar, el resultado es análogo en todas las conectivas binarias aunque aparezcan definidas por separado, por tanto haré la demostración estructurada para las definiciones simplificadas. Nos basaremos en el lema anterior *subsubformulae*.

lemma *subformulas-in-subformulas-estructurada1-s*:
assumes *Bi G H* $\in$ *setSubformulae-s F*
shows $G \in$ *setSubformulae-s F* $\wedge\ H \in$ *setSubformulae-s F*
proof (*rule conjI*)
have *1*:*setSubformulae-s* (*Bi G H*) = {*Bi G H*} $\cup$ *setSubformulae-s G* $\cup$ *setSubformulae-s H* **by** *simp*
then have *2*:$G \in$ *setSubformulae-s* (*Bi G H*) **by** *simp*
have *3*:$H \in$ *setSubformulae-s* (*Bi G H*) **using** *1* **by** *simp*
show $G \in$ *setSubformulae-s F* **using** *assms 2* **by** (*rule subsubformulae-s*)
show $H \in$ *setSubformulae-s F* **using** *assms 3* **by** (*rule subsubformulae-s*)
qed

lemma *subformulas-in-subformulas-negacion-estructurada*:
assumes *Mon G* $\in$ *setSubformulae-s F*
shows $G \in$ *setSubformulae-s F*
proof −
have *setSubformulae-s* (*Mon G*) = {*Mon G*} $\cup$ *setSubformulae-s G* **by** *simp*
then have *1*:$G \in$ *setSubformulae-s* (*Mon G*) **by** *simp*
show $G \in$ *setSubformulae-s F* **using** *assms 1* **by** (*rule subsubformulae-s*)
qed

Mostremos ahora la demostración aplicativa para estos casos y la automática para el lema completo.

lemma *subformulas-in-subformulas-aplicativa-s*:
Bi G H $\in$ *setSubformulae-s F* $\Longrightarrow G \in$ *setSubformulae-s F* $\wedge\ H \in$ *setSubformulae-s F*

Mon G ∈ *setSubformulae-s F* ⟹ *G* ∈ *setSubformulae-s F*
apply (*rule conjI*)
apply (*erule subsubformulae-s*,*simp*)+
done

lemma *subformulas-in-subformulas*:
G ∧ *H* ∈ *setSubformulae F* ⟹ *G* ∈ *setSubformulae F* ∧ *H* ∈ *setSubformulae F*
G ∨ *H* ∈ *setSubformulae F* ⟹ *G* ∈ *setSubformulae F* ∧ *H* ∈ *setSubformulae F*
G ⟶ *H* ∈ *setSubformulae F* ⟹ *G* ∈ *setSubformulae F* ∧ *H* ∈ *setSubformulae F*
¬ *G* ∈ *setSubformulae F* ⟹ *G* ∈ *setSubformulae F*
by (*fastforce elim*: *subsubformulae*)+

Concluimos la sección de subfórmulas con un resultado que relaciona varias funciones sobre la longitud de la lista *subformulae F* de una fórmula *F* cualquiera.

lemma *length-subformulae*: *length* (*subformulae F*) = *size F*
oops

En prime lugar aparece la clase *size* de la teoría de números naturales .... Vamos a definir *size1* de manera idéntica a como aparace *size* en la teoría.

class *size1* =
fixes *size1* :: *'a* ⇒ *nat*

instantiation *nat* :: *size1*
begin

definition *size1-nat* **where** [*simp*, *code*]: *size1* (*n*::*nat*) = *n*

instance ..

end

Como podemos observar, se trata de una clase que actúa sobre un parámetro global de tipo *'a* cualquiera. Por otro lado, *instantation* define una clase de parámetros, en este caso los números naturales *nat* que devuelve como resultado. Incluye una definición concreta del operador *size1* sobre dichos parámetros. Además, el último *instance* abre una prueba que afirma que los parámetros dados conforman la clase especificada en la definición.

Esta prueba que nos afirma que está bien definida la clase aparece omitida utilizando .. .
En particular, sobre una fórmula nos devuelve el número de elementos de la lista cuyos elementos son los nodos y las hojas de su árbol de formación.

value $size(n::nat) = n$
value$size(5::nat) = 5$

Por otro lado, la función *length* de la teoría List.thy nos indica la longitud de una lista cualquiera de elementos, definiéndose utilizando el comando *size* visto anteriormente.

abbreviation $length :: 'a\ list \Rightarrow nat$ **where**
$length \equiv size$

Las demostración del resultado se vuelve a basar en la inducción que nos despliega seis casos. La prueba estructurada no resulta interesante, pues todos los casos son inmediatos por simplificación como en el primer lema de esta sección. Incluimos a continuación la prueba aplicativa y automática.

lemma *length-subformulae-aplicativa*: $length\ (subformulae\ F) = size\ F$
  apply (*induction F*)
  apply *simp-all*
 done

lemma *length-subformulae*: $length\ (subformulae\ F) = size\ F$
  by (*induction F*; *simp*)

1.2 Conectivas Derivadas

En esta sección definiremos nuevas conectivas y elementos a partir de los ya definidos en el apartado anterior. Además veremos varios resultados sobre los mismos.

En primer lugar, vamos a definir $Top:: 'a\ formula \Rightarrow bool$ como la constante que devuelve el booleano contrario a *Bot*. Se trata, por tanto, de una constante de la misma naturaleza que la ya definida para *Bot*. De este modo, *Top* será equivalente a *Verdadero*, y *Bot* a *Falso*, según se muestra en la siguiente ecuación. Su símbolo queda igualmente retratado a continuación.

definition *Top* ($\top$) **where**
$\top \equiv \bot \rightarrow \bot$

Por la propia definición y naturaleza de *Top*, verifica que no contiene ninguna variable del alfabeto, como ya sabíamos análogamente para *Bot*. Tenemos así la siguiente propiedad.

lemma *top-atoms-simp*[*simp*]: *atoms* $\top = \{\}$
  unfolding *Top-def* **by** *simp*

A continuación vamos a definir dos conectivas que generalizarán la conjunción y la disyunción para una lista finita de fórmulas. .

primrec *BigAnd* :: *'a formula list* $\Rightarrow$ *'a formula* ($\bigwedge$-) **where**
  $\bigwedge Nil = (\neg\bot)$
| $\bigwedge(F\#Fs) = F \wedge \bigwedge Fs$

primrec *BigOr* :: *'a formula list* $\Rightarrow$ *'a formula* ($\bigvee$-) **where**
  $\bigvee Nil = \bot$
| $\bigvee(F\#Fs) = F \vee \bigvee Fs$

Ambas nuevas conectivas se caracterizarán por ser del tipo funciones primitivas recursivas. Por tanto, sus definiciones se basan en dos casos:

Lista vacía: Representada como *Nil*. En este caso, la conjunción plural aplicada a la lista vacía nos devuelve la negación de *Bot*, es decir, *Verdadero*, y la disyunción plural sobre la lista vacía nos da simplemente *Bot*, luego *Falso*.

Lista recursiva: En este caso actúa sobre $F\#Fs$ donde F es una fórmula concatenada a la lista de fórmulas *Fs*. Como es lógico, *BigAnd* nos devuelve la conjunción de todas las fórmulas de la lista y *BigOr* nos devuelve su disyunción.

Además, se observa en cada función el símbolo de notación que aparece entre paréntesis. La conjunción plural nos da el siguiente resultado.

lemma *atoms-BigAnd*[*simp*]: *atoms* $(\bigwedge Fs) = \bigcup(atoms \text{ ` } set\ Fs)$
  by(*induction Fs*; *simp*)

Referencias

[1] José A. Alonso. Temas de "Lógica matemática y fundamentos (2018–19)". Technical report, Univ. de Sevilla, 2019. En https://www.cs.us.es/~jalonso/cursos/lmf-18/temas.php.

[2] M. Fitting. *First-order Logic and Automated Theorem Proving*. Graduate texts in computer science. Springer, 1996.

[3] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for Higher–Order Logic*. Lecture Notes in Computer Science, Vol. 2283, Springer–Verlag, 2019. En https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2019/doc/tutorial.pdf.