Capítulo 1

Isabelle como un lenguaje funcional

1.1 Introducción

Nota 1.1.1. Esta notas son una introducción a la demostración asistida utilizando el sistema Isabelle/HOL/Isar. La versión de Isabelle utilizada es la 2009-2.

Nota 1.1.2. Un **lema** introduce una proposición seguida de una demostración. Isabelle dispone de varios procedimientos automáticos para generar demostraciones, uno de los cuales es el de simplificación (llamado *simp*). El procedimiento *simp* aplica un conjunto de reglas de reescritura que inicialmente contiene un gran número de reglas relativas a los objetos definidos. El ejemplo del lema más trivial es el siguiente

lemma *elMasTrivial*: *True*
by *simp*

En este capítulos se presenta el lenguaje funcional que está incluido en Isabelle. El lenguaje funcional es muy parecido al ML estándard.

1.2 Números naturales, enteros y booleanos

Nota 1.2.1 (Números naturales)**.**

- En Isabelle están definidos los número naturales con la sintaxis de Peano usando dos constructores: *0* (cero) y *Suc n* (el sucesor de *n*).
- Los números como el *1* son abreviaturas de los correspondientes en la notación de Peano, en este caso *Suc 0*.
- El tipo de los números naturales es *nat*.

Lema 1.2.2 (Ejemplo de simplificación de números naturales)**.** *El siguiente del* 0 *es el* 1*.*

lemma *Suc 0 = 1*
by *simp*

Nota 1.2.3 (Suma y producto de números naturales)**.** En Isabelle están definida la suma y el producto de números naturales:

- $x+y$ es la suma de x e y
- $x*y$ es el producto de x e y

Lema 1.2.4 (Ejemplo de suma)**.** *La suma de los números naturales 1 y 2 es el número natural 3.*

lemma $1 + 2 = (3::nat)$
by *simp*

Nota 1.2.5 (Epecificación de tipo)**.** La notación del par de dos puntos se usa para asignar un tipo a un término (por ejemplo, `3::nat` significa que se considera que `3` es un número natural).

Lema 1.2.6 (Ejemplo de producto)**.** *El producto de los números naturales 2 y 3 es el número natural 6.*

lemma $2 * 3 = (6::nat)$
by *simp*

Nota 1.2.7 (División de números naturales)**.** En Isabelle está definida la división de números naturales: *n div m* es el mayor número natural que multiplicado por m es menor o igual que n.

Lema 1.2.8 (Ejemplo de división)**.** *La división natural de 7 entre 3 es 2.*

lemma *7 div 3* $= (2::nat)$
by *simp*

Nota 1.2.9 (Resto de división de números naturales)**.** En Isabelle está definida el resto de división de números naturales: *n mod m* es el resto de dividir n entre m.

Lema 1.2.10 (Ejemplo de resto)**.** *El resto de dividir 7 entre 3 es 1.*

lemma *7 mod 3* $= (1::nat)$
by *simp*

Nota 1.2.11 (Números enteros)**.** En Isabelle también están definidos los números enteros. El tipo de los enteros se representa por *int*.

Lema 1.2.12 (Ejemplo de operación con enteros)**.** *La suma de 1 y -2 es el número entero -1.*

lemma $1 + -2 = (-1::int)$
by *simp*

Nota 1.2.13 (Sobrecarga)**.** Los numerales están sobrecargados. Por ejemplo, el '1' puede ser un natural o un entero, dependiendo del contexto. Isabelle resuelve ambigüedades mediante inferencia de tipos. A veces, es necesario usar declaraciones de tipo para resolver la ambigüedad.

Nota 1.2.14 (Booleanos, conectivas y cuantificadores)**.** En Isabelle están definidos los valores booleanos *True*, *False*, las conectivas $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\leftrightarrow$ y los cuantificadores $\forall$, $\exists$. El tipo de los booleanos es *bool*.

Lema 1.2.15 (Ejemplos de evaluaciones booleanas)**.**

1. *La conjunción de dos fórmulas verdaderas es verdadera.*
2. *La conjunción de un fórmula verdadera y una falsa es falsa.*
3. *La disyunción de una fórmula verdadera y una falsa es verdadera.*
4. *La disyunción de dos fórmulas falsas es falsa.*
5. *La negación de una fórmula verdadera es falsa.*
6. *Una fórmula falsa implica una fórmula verdadera.*
7. *Todo elemento es igual a sí mismo.*
8. *Existe un elemento igual a 1.*

lemma $True \wedge True = True$
by *simp*

lemma $True \wedge False = False$
by *simp*

lemma $True \vee False = True$
by *simp*

lemma $False \vee False = False$

by *simp*

lemma $\neg$ *True* = (*False*::*bool*)
by *simp*

lemma *False* $\longrightarrow$ *True*
by *simp*

lemma $\forall$ *x*. *x* = *x*
by *simp*

lemma $\exists$ *x*. *x* = *1*
by *simp*

1.3 Definiciones no recursivas

Definición 1.3.1 (Ejemplo de definición no recursiva)**.** *La disyunción exclusiva de A y B se verifica si una es verdadera y la otra no lo es.*

definition *xor* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where**
xor A B $\equiv$ (*A* $\wedge$ $\neg$ *B*) $\vee$ ($\neg$ *A* $\wedge$ *B*)

Lema 1.3.2 (Ejemplo de demostración con definiciones no recursivas)**.** *La disyunción exclusiva de dos fórmulas verdaderas es falsa.*

Demostración: Por simplificación, usando la definición de la disyunción exclusiva.
□

lemma *xor True True* = *False*
by (*simp add*: *xor-def*)

Nota 1.3.3 (Ejemplo de ampliación de las reglas de simplificación)**.** Se añade la definición de la disyunción exlusiva al conjunto de reglas de simplificación automáticas.

declare *xor-def*[*simp*]

1.4 Definiciones locales

Nota 1.4.1 (Variables locales)**.** Se puede asignar valores a variables locales mediante 'let' y usarlo en las expresiones dentro de 'in'.

Lema 1.4.2 (Ejemplo de entorno local)**.** *Sea x el número natural 3. Entonces* $x \times x = 9$.

lemma (*let* $x = 3$::*nat in* $x * x = 9$)
by *simp*

1.5 Pares

Nota 1.5.1 (Pares)**.**

- Un par se representa escribiendo los elementos entre paréntesis y separados por coma.
- El tipo de los pares es el producto de los tipos.
- La función *fst* devuelve el primer elemento de un par y la *snd* el segundo.

Lema 1.5.2 (Ejemplo de uso de pares)**.** *Sea p el par de números naturales* $(2,3)$. *La suma del primer elemento de p y* 1 *es igual al segundo elemento de p.*

lemma *let* $p = (2,3)$::*nat* $\times$ *nat in fst p* $+ 1 =$ *snd p*
by *simp*

1.6 Listas

Nota 1.6.1 (Construcción de listas)**.**

- Una lista se representa escribiendo los elementos entre corchetes y separados por coma.
- La lista vacía se representa por `[]`.
- Todos los elementos de una lista tienen que ser del mismo tipo.
- El tipo de las listas de elementos del tipo `a` es `a list`.
- El término `a#l` representa la lista obtenida añadiendo el elemento `a` al principio de la lista `l`.

Lema 1.6.2 (Ejemplo de construcción de listas)**.** *La lista obtenida añadiendo sucesivamente a la lista vacía los elementos 3, 2 y 1 es [1,2,3].*

lemma *1*#(*2*#(*3*#[])) = [*1,2,3*]
by *simp*

Nota 1.6.3 (Primero y resto)**.**

- `hd l` es el primer elemento de la lista `l`.
- `tl l` es el resto de la lista `l`.

Lema 1.6.4 (Ejemplo de cálculo con listas)**.** *Sea* `l` *la lista de números naturales* `[1,2,3]`*. Entonces, el primero de* `l` *es 1 y el resto de* `l` *es* `[2,3]`*.*

lemma *let* $l = [1,2,3]::(nat\ list)$ *in hd* $l = 1 \wedge tl\ l = [2,3]$
by *simp*

Nota 1.6.5 (Longitud)**.** `length l` es la longitud de la lista `l`.

Lema 1.6.6 (Ejemplo de cálculo de longitud)**.** *La longitud de la lista [1,2,3] es 3.*

lemma *length* $[1,2,3] = 3$
by *simp*

Nota 1.6.7 (Referencias sobre listas)**.** En la sesión 38 de "HOL: The basis of Higher-Order Logic" se encuentran más definiciones y propiedades de las listas.

1.7 Registros

Nota 1.7.1 (Registro)**.** Un registro es una colección de campos y valores.

Definición 1.7.2 (Ejemplo de definición de registro)**.** *Los puntos del plano pueden representarse mediante registros con dos campos, las coordenadas, con valores enteros.*

record *punto* =
 coordenada-x :: *int*
 coordenada-y :: *int*

Definición 1.7.3 (Ejemplo de definición de un registro)**.** *El punto* `pt` *tiene de coordenadas 3 y 7.*

definition *pt* :: *punto* **where**
 $pt \equiv (\lfloor coordenada\text{-}x = 3, coordenada\text{-}y = 7 \rfloor)$

Lema 1.7.4 (Ejemplo de propiedad de registro)**.** *La coordenada x del punto* `pt` *es 3.*

lemma *coordenada-x pt* = *3*
by (*simp add*: *pt-def*)

Lema 1.7.5 (Ejemplo de actualización de un registro)**.** *Sea* `pt2` *el punto obtenido a partir del punto* `pt` *cambiando el valor de su coordenada x por 4. Entonces la coordenada x del punto* `pt2` *es 4.*

lemma *let pt2*=*pt*(|*coordenada-x*:=*4*|) *in coordenada-x* (*pt2*) = *4*
by (*simp add*: *pt-def*)

1.8 Funciones anónimas

Nota 1.8.1 (Funciones anónimas)**.** En Isabelle pueden definirse funciones anónimas.

Lema 1.8.2 (Ejemplo de uso de funciones anónimas)**.** *El valor de la función que a un número le asigna su doble aplicada a 1 es 2.*

lemma (λ *x*. *x* + *x*) *1* = (*2*::*nat*)
by *simp*

1.9 Condicionales

Definición 1.9.1 (Ejemplo con el condicional *if*)**.** *El valor absoluto del entero x es x, si x* ≥ *0 y es* −*x en caso contrario.*

definition *absoluto* :: *int* ⇒ *int* **where**
absoluto x ≡ (*if x* ≥ *0 then x else* −*x*)

Lema 1.9.2 (Ejemplo de simplificación con el condicional *if*)**.** *El valor absoluto de -3 es 3.*

lemma *absoluto*(−*3*) = *3*
by (*simp add*:*absoluto-def*)

Definición 1.9.3 (Ejemplo con el condicional *case*)**.** *Un número natural n es un sucesor si es de la forma Suc m.*

definition *es-sucesor* :: *nat* ⇒ *bool* **where**
es-sucesor n ≡
(*case n of*
0 ⇒ *False*

| *Suc m* ⇒ *True*)

Lema 1.9.4 (Ejemplo de simplificación con el condicional *case*)**.** *El número 3 es sucesor.*

lemma *es-sucesor 3*
by (*simp add*: *es-sucesor-def*)

1.10 Tipos de datos y recursión primitiva

Definición 1.10.1 (Ejemplo de definición de tipo de dato recursivo)**.** *Una lista de elementos de tipo* `a` *es la lista* `Vacia` *o se obtiene añadiendo, con* `ConsLista`*, un elemento de tipo* `a` *a una lista de elementos de tipo* `a`*.*

datatype *'a Lista* = *Vacia* | *ConsLista 'a 'a Lista*

Definición 1.10.2 (Ejemplo de definición primitiva recursiva)**.** `conc xs ys` *es la concatenación de las lista* `xs` *e* `ys`*.*

primrec *conc* :: *'a Lista* ⇒ *'a Lista* ⇒ *'a Lista* **where**
conc Vacia ys = *ys*
| *conc* (*ConsLista x xs*) *ys* = *ConsLista x* (*conc xs ys*)

Lema 1.10.3 (Ejemplo de simplificación con tipo de dato recursivo)**.** *La concatenación de la lista formada por 1 y 2 con la lista formada por el 3 es la lista cuyos elementos son 1,2 y 3.*

lemma *conc* (*ConsLista 1* (*ConsLista 2 Vacia*)) (*ConsLista 3 Vacia*) =
(*ConsLista 1* (*ConsLista 2* (*ConsLista 3 Vacia*)))
by *simp*

Ejercicio 1.10.4 (Ejemplo de definición primitiva recursiva sobre los naturales)**.** Definir una función que sume los primeros n números naturales y usarla para comprobar que la suma de los 3 primeros números naturales es 6.

primrec *suma* :: *nat* ⇒ *nat* **where**
suma 0 = *0*
| *suma* (*Suc m*) = (*Suc m*) + *suma m*

lemma *suma 3* = *6*
by (*simp add*: *suma-def*)