Análisis Formal de Conceptos desde el punto de vista de la programación funcional

Realizado por:
María Najarro Gómez

Junio de 2016

Trabajo dirigido por:
María José Hidalgo Doblado y
José Antonio Alonso Jiménez

Dpto. de Ciencias de la Computación
e Inteligencia Artificial
Facultad de Matemáticas
Universidad de Sevilla

Índice general

1. Introducción **3**

2. Introducción a Haskell **7**
2.1. ¿Qué es Haskell? 7
2.2. Características esenciales 8
2.3. Diferencias con otros lenguajes de programación 10
2.4. ¿Por qué se ha elegido Haskell? 11
2.5. Utilidades importantes de Haskell 12

3. El retículo de los conceptos de un contexto formal **23**
3.1. Contextos formales 23
3.1.1. Representación del contexto formal como TAD 24
3.1.2. Propiedades del contexto formal 33
3.2. Conceptos formales 47
3.2.1. Definiciones 47
3.2.2. Retículo de los conceptos 48
3.2.3. Generación de todos los conceptos de un contexto formal 54

4. Razonamiento en contextos formales **61**
4.1. Implicaciones entre atributos 61
4.2. Base de implicaciones 66
4.3. Cálculo de la base Stem 67

Referencias **75**

Apéndice **77**

Abstract

Formal Concept Analysis is a mathematical theory of data analysis with growing popularity across various domains such as psichology, biology, mathematics, medicine or industrial engineering. Formal Concept Analysis analyzes data wich describe relationship between a particular set of objects and a particular set of attributes (properties of objects). The original motivation of Formal Concept Analysis was the concrete representation of complete lattices and their properties by means of formal contexts. Formal contexts are data tables that represent binary relations between objects and attributes. This theory has an special importance when it is necesary to work with large set of objects which can be described by a large set of properties or atributes.

This work is concerned with an implementation of fundamental concepts and methods of formal concept analysis in a functional programming language. We choose Haskell for this task and we follow the theory which appears in the book "Formal Concept Analysis" [1].

Capítulo 1

Introducción

El **Análisis Formal de Conceptos**[1] es una teoría matemática y una técnica de aprendizaje, de análisis de datos, de representación del conocimiento y manejo de información que proporciona una base para realizar una clasificación conceptual de la información.

El término fue introducido por *Rudolf Wille* en 1984, y se basa en la Teoría de Retículos y en la Teoría de Órdenes, que fue desarrollada por *Garrett Birkhoff* y otros en la década de los años 30.

En los primeros diez años, fue desarrollado principalmente por un pequeño grupo de investigadores y estudiantes. Con el tiempo, el AFC fue implementado en varias aplicaciones de gran escala, siendo la mayoría una implementación de sistemas de exploración de conocimiento para ingeniería civil.

No obstante, durante los últimos años, ha evolucionado en gran medida con aplicaciones no sólo en dominios tecnológicos como la Informática, sino que también ha sido aplicado con éxito en otras disciplinas más descriptivas y variadas, tales como la psicología, la sociología, la antropología, la biología, las matemáticas, la medicina, la lingüística o la ingeniería industrial. Como ejemplos, enumeramos algunas de sus aplicaciones:

- Clasificación y procesamiento de correos electrónicos.
- Descubrimiento de conocimiento en textos médicos.
- Técnicas de aprendizaje.
- Análisis de herencia en una jerarquía de clases.

[1] Llamaremos AFC al Análisis Formal de Conceptos en todo lo que sigue el texto.

El estudio del AFC y de otras técnicas similares tiene especial interés cuando es necesario trabajar con un gran número de entidades u objetos que pueden describirse mediante un amplio conjunto de propiedades o atributos. Estas técnicas permiten la clasificación y estructuración de toda esta información basando la jerarquía obtenida en los conceptos formales del dominio, que son pares de conjuntos de objetos y atributos.

El objetivo de este trabajo es la implementación de parte de la teoría del AFC en el lenguaje de programación Haskell. Para dicha teoría nos hemos guiado del libro *Formal Concept Analysis* [1] donde se explica con detalle cada aspecto del Análisis Formal de Conceptos. Aunque existe una amplia variedad de software desarrollados para trabajar con AFC, como puede verse en [13], en este trabajo nos proponemos comenzar la implementación de esta teoría en un lenguaje de programación funcional como Haskell, que es el que hemos utilizado en las asignaturas del grado. En lo que conocemos hasta ahora, el AFC sólo ha sido implementado en Haskell por Thomas Sutton [14] [15]. El objetivo principal de su trabajo era dibujar el retículo de los conceptos.

El propósito del trabajo que presentamos es comenzar el desarrollo de la teoría del AFC en Haskell. En este trabajo:

- Hemos representado en Haskell los contextos formales usando tipos abstractos de datos.
- Hemos implementado el retículo de los conceptos de un contexto formal.
- Hemos implementado algoritmos para generar todos los conceptos de un contexto.
- Hemos representado la noción de implicación válida en un contexto.
- Finalmente, hemos implementado un algoritmo para generar una base de implicaciones, llamada **Base de Duquenne-Guigues**.

Además, hemos usado QuickCheck [12] para especificar y comprobar mediante una batería de ejemplos las propiedades matemáticas de la teoría.

Resumimos, a continuación, el contenido de cada capítulo.

En el *segundo capítulo* trataremos sobre una introducción al lenguaje Haskell. Para esta tarea hemos utilizado algunas páginas web como son la página oficial de Haskell, *HaskellWiki* [3] y *¡Aprende Haskell por el bien de todos!* [4]. A lo largo del capítulo se incluyen las principales características

de este lenguaje, además de una enumeración de pros y contras frente a otros lenguajes de programación; hecho que nos lleva a la utilización de este lenguaje para la realización del trabajo que aquí presentamos. Concluiremos este capítulo describiendo una serie de utilidades y herramientas importantes de este programa, como por ejemplo QuickCheck y los Tipos Abstractos de Datos, que utilizaremos a lo largo del grueso del trabajo.

Iniciaremos el *tercer capítulo* con las primeras definiciones de la teoría del AFC y sus respectivas implementaciones en Haskell, además de las diferentes representaciones de los Tipos Abstractos de Datos que usaremos en lo que sigue. Comenzaremos definiendo los contextos formales y daremos a conocer una serie de propiedades que estos contextos cumplen. Una vez tengamos estas nociones básicas, pasaremos a definir los conceptos formales y algunas características que nos premitirán estructurar el retículo de los conceptos. Por último, concluiremos el capítulo con varios algoritmos para la generación de todos los conceptos de un contexto formal.

El *capítulo cuarto* trata sobre las relaciones entre los atributos de un contexto formal, para lo que definiremos la noción de implicación. La finalidad de este capítulo es el cálculo de la base Stem de un contexto dado. Para ello buscaremos un conjunto de implicaciones cumpliendo una serie de propiedades: adecuación, completitud y redundancia; para ello, definiremos la noción de implicación válida en un contexto y la noción de pseudo-intensión.

Capítulo 2

Introducción a Haskell

2.1. ¿Qué es Haskell?

Haskell es un lenguaje de programación puramente funcional que debe su nombre al lógico estadounidense Haskell Brooks Curry. Haskell está basado en el lambda cálculo[1] y es por esto por lo que el símbolo lambda es usado como logo. Este lenguaje de programación surge debido a que, a partir de la publicación del programa Miranda en 1985, los lenguajes funcionales proliferaron. La primera versión de Haskell ("Haskell 1.0") se desarrolló en 1990. Más tarde se desarrollaron una serie de definiciones del lenguaje, que culminaron a finales de 1997 en "Haskell 98"; que se intentó fuera una versión del lenguaje mínima, estable y portable, junto con una biblioteca estándar asociada para la enseñanza, y como base de futuras extensiones. El lenguaje progresa y en 2010 se lanza "Haskell 2010". A día de hoy, Haskell continúa en evolución.

Es importante mencionar que Haskell está siendo usado tanto en investigación, como en educación y en industria[2]. Algunos de los ejemplos de su uso en educación e industria son los siguientes:

- Educación:
    - Como primer lenguaje de programación en universidades como la de Sevilla; en la asignatura de "Informática", y otras como la

[1] El **lambda cálculo** es el lenguaje universal de programación más pequeño que existe. Consiste en una regla de transformación simple (sustituir variables) y un esquema simple para definir funciones.

[2] Más detalle en su página oficial [3].

de Edimburgo; en la asignatura "Informática 1 - Programación funcional".

- Como segundo lenguaje de programación. Por ejemplo en la Universidad de Oklahoma en su asignatura de "Matemáticas Discretas".

- Industria:
  - En la compañía de servicios financieros *Barclays Capital Quantitative Analytics Group*.
  - En entidades bancarias como *Deutsche Bank Equity Proprietary Trading, Directional Credit Trading*.
  - En *Facebook*.

2.2. Características esenciales

Las principales características del lenguaje Haskell son las siguientes:

- Está especialmente diseñado para manejar una amplia **gama de aplicaciones**, desde análisis numérico hasta simbólico. Para alcanzar estos objetivos, Haskell posee una sintaxis expresiva y una rica variedad de tipos primitivos; incluyendo enteros y racionales de precisión arbitraria, además de los tipos de enteros, punto flotante y booleanos. También otros como listas, caracteres y cadenas.

- Por ser lenguaje puramente funcional, una función no tiene efectos secundarios. Lo que hace una función es calcular y devolver algo como resultado. Si una función es llamada dos veces con los mismos parámetros, obtendremos siempre el mismo resultado. A esto lo llamamos **transparencia referencial**. En otras palabras, Haskell es **puro** respecto de la integridad referencial; no permite ningún efecto colateral. Probablemente, sea una de sus características más importantes.

- Haskell posee **evaluación perezosa**; dicho más técnicamente, es **no estricto**. Es decir, a menos que le indiquemos lo contrario, Haskell no ejecutará funciones ni calculará resultados hasta que se vea realmente forzado a hacerlo. Esto funciona muy bien junto con la transparencia

referencial y permite que veamos los programas como una serie de transformaciones de datos. Incluso nos permite trabajar con estructuras de datos infinitas. Así, por ejemplo, se puede definir una lista infinita de primos. Sólo los elementos de esta lista que sean realmente usados serán construidos en ese momento. De hecho, en este caso se debe ver la definición como un patrón de generación de los elementos que contiene, no como el conjunto en sí mismo, lo que permite algunas soluciones muy elegantes para muchos problemas. De este modo, el ordenador sólo hace un recorrido a través de la lista y sólo cuando lo necesitamos.

- Haskell es un lenguaje **fuertemente tipado**[3]. Cuando compilamos un programa, el compilador sabe qué trozos del código son enteros, cuáles son cadenas de texto, etc. Gracias a esto podemos detectar gran cantidad de posibles errores en el tiempo de compilación. Por ejemplo, si intentamos sumar un número y una cadena de texto, el compilador nos dará error. Haskell no es el único lenguaje fuertemente tipado, pero a diferencia de la mayoría de ellos, en Haskell estos tipos son inferidos automáticamente, lo que quiere decir que no tenemos que etiquetar cada trozo de código explícitamente con un tipo porque el sistema de tipos lo puede deducir de forma inteligente. La inferencia de tipos también permite que nuestro código sea más general, si hemos creado una función que toma dos números y los suma y no establecemos explícitamente sus tipos, la función aceptará cualquier par de parámetros que actúen como números.

- Haskell es **elegante** y **conciso**. Esto se debe a que permite la programación de alto nivel. Los programas Haskell son normalmente más cortos que los equivalentes imperativos. Y los programas cortos son más fáciles de mantener que los largos, además de que poseen menos errores.

- Haskell es **modular**; es decir, para resolver un problema, podemos dividirlo en problemas más pequeños. De esta manera, en lugar de resolver una tarea compleja y tediosa, resolvemos otras más sencillas y a partir de ellas llegamos a la solución.

[3]Se dice que un lenguaje de programación es **fuertemente tipado** cuando la comprobación de los tipos se realiza durante el tiempo de compilación.

- **Veloz**. Aunque algunos programadores consideran otros lenguajes más rápidos que Haskell, existen comparativas que muestran que Haskell no es tan lento como algunas personas piensan. Actualmente, está de media en la segunda posición, sólo ligeramente detrás de `C`.

- Buena **gestión de memoria**. No hay que preocuparse por la memoria, el Garbage Collector[4] de Haskell se ocupa de todo, y lo hace de una forma extremadamente eficiente. Nos permite preocuparnos únicamente de la implementación del algoritmo, no de cómo gestionar la memoria.

2.3. Diferencias con otros lenguajes de programación

En los lenguajes imperativos obtenemos resultados dándole al ordenador una secuencia de tareas que luego éste ejecutará. Mientras las ejecuta, puede cambiar de estado. Por ejemplo, establecemos la variable x a 5, realizamos algunas tareas y luego cambiamos el valor de la variable anterior. Estos lenguajes poseen estructuras de control de flujo para realizar ciertas acciones varias veces (`for`, `while`...). Con la programación puramente funcional no decimos al ordenador lo que tiene que hacer, sino más bien, decimos cómo son las cosas. El factorial de un número es el producto de todos los números desde el 1 hasta ese número, la suma de una lista de números es el primer número más la suma del resto de la lista, etc. En definitiva, expresamos la forma de las funciones.

Además los programas funcionales tienden a ser mucho más concisos que sus correspondientes imperativos.

Aún así, Haskell presenta algunos inconvenientes como:

- Mayor dificultad inicial, pues aunque sean muy fáciles de entender y mantener, suele ser más difícil escribir un programa funcionalmente, sobre todo para mentes acostumbradas a lo imperativo. La ausencia de variables de estado hace que tengamos que planificar muy bien lo que vamos a hacer.

[4]Un recolector de basura (del inglés **garbage collector**) es un mecanismo implícito de gestión de memoria implementado en algunos lenguajes de programación.

- Posible falta de recursos al estar menos extendido que otros. A pesar de que tiene una gran cantidad de librerías enumeradas por categorías[5],como pueden ser *Cryptography*, *Financial* o *Robotics*, otros programas tienen incluso más librerías disponibles.

A pesar de estos contras pesan más los pros, pues son muchas las características que nos aportan un clima adecuado de trabajo. Además de las esenciales, recogidas en el punto anterior, cabe destacar su sencilla instalación ya que sólo se necesita un editor de texto y un compilador de Haskell.

2.4. ¿Por qué se ha elegido Haskell?

En primer lugar elegí Haskell porque fue el lenguaje de programación sugerido por mis tutores de TFG. La idea no me desagradó pues es un lenguaje que me resulta familiar ya que con él inicié mis conocimientos de programación en la materia de Informática de primer curso. En segundo lugar, me ayuda a poner en práctica todo lo aprendido de Haskell durante mis años de carrera además de afianzar conocimientos y aprender cosas nuevas.

Además se adecúa fácilmente a la representación de conceptos matemáticos, que es el objetivo del trabajo. Un ejemplo de este hecho es la posibilidad de definir listas por comprensión que se asemeja a la definición matemática de conjuntos de elementos.

Por otro lado, porque al ser un lenguaje de programación funcional tiene muchas ventajas frente a los lenguajes imperativos. Escribir programas grandes que funcionen correctamente es difícil y costoso. Mantener esos programas es aún más difícil y costoso. Los lenguajes de programación funcional pueden hacerlo mucho más fácil.

Además, este tipo de programas son también relativamente fáciles de mantener porque el código es más corto, claro y el riguroso control sobre efectos colaterales elimina una amplia clase de interacciones imprevisibles.

Incluso si no se está familiarizado con Haskell, aprenderlo puede hacernos un mejor programador en cualquier otro lenguaje.

Haskell ofrece:

- Un código más corto, claro y fácil de mantener.

[5]Podemos encontrar todas las librerías de Haskell en la siguiente página: `https://hackage.haskell.org/packages/`

- Menos errores y, por tanto, mayor confiabilidad.

- Una menor diferencia semántica entre el programador y el lenguaje.

- Un desarrollo de programas en menor tiempo.

Haskell es un lenguaje de un amplio espectro, conveniente para una gran variedad de aplicaciones. Específicamente para programas que necesitan ser fáciles de modificar y de mantener.

Ha de mencionarse que se ha convertido en el gran representante de la programación funcional, en el referente de todo lo que un lenguaje funcional puro debe y puede ser.

2.5. Utilidades importantes de Haskell

Haskell posee una herramienta llamada **QuickCheck** que nos ayuda a formular y comprobar que se verifican ciertas propiedades de las funciones que componen un programa. Estas propiedades se expresan como otras funciones de Haskell y pueden ser comprobadas automáticamente con entradas aleatorias.

Las comprobaciones aleatorias son especialmente adecuadas para programas funcionales porque las propiedades se pueden expresar con bastante granularidad; está generalmente aceptado que las propiedades de las funciones puras son mucho más fáciles de verificar que las que tienen efectos secundarios porque uno no tiene que preocuparse por el estado antes y después de su ejecución. Las herramientas de comprobación automática facilitan completar la realización de dichos chequeos en menor tiempo o una comprobación con mayor rigor en el mismo tiempo disponible y hacen que sea fácil repetir dichas comprobaciones después de cada modificación del programa.

QuickCheck comprueba entonces que las propiedades se cumplen para un número grande de casos. Es también una herramienta para realizar comprobaciones capaces de generar casos de prueba automáticamente.

Debe quedar claro que QuickCheck no es un demostrador de propiedades, como es el caso de otros programas como Isabelle/HoL.

Resumamos su funcionamiento:

1. Definimos una propiedad en Haskell. Hay que tener en cuenta que se pueden distinguir propiedades de dos tipos:

- Propiedades que son simples ecuaciones representadas convenientemente por funciones booleanas. Por ejemplo, la propiedad "el doble de x más y es el doble de x más el doble de y" la podemos representar por:

```
prop_doble :: Int -> Int -> Bool
prop_doble x y =
        doble (x+y) == (doble x) + (doble y)
                where doble x = x + x
```

- Propiedades que sólo se cumplen bajo ciertas condiciones. Por ejemplo, la ley "x menor o igual que y implica que el máximo entre x e y es igual a y" puede ser representada por esta definición:

```
prop_Max :: Int -> Int -> Property
prop_Max x y = x <= y ==> max x y == y
```

  Podemos ver que para esta propiedad, el tipo no es `Bool` sino `Property`. Esto es así porque la semántica es distinta; si el lado izquierdo de la implicación no es `True`, se descarta ese caso y se intenta uno nuevo.

2. Compilamos nuestro programa y comprobamos si la propiedad es correcta escribiendo `quickCheck` seguido del nombre de dicha propiedad. Por defecto, Haskell crea 100 ejemplos aleatorios (del tipo especificado en la función) para comprobar la propiedad. Aún así, se puede modificar para que, en vez de 100, genere más o menos ejemplos. En algunos casos, como por ejemplo cuando los datos de entrada son listas, podemos exigirle que el tamaño de las listas que generará Haskell de manera aleatoria, no exceda de unas dimensiones específicas. Ésto lo hacemos con la orden `quickCheckWith (stdArgs maxSize=n)`, donde n sería la dimensión máxima de los ejemplos que genera.

3. Una vez haga la comprobación pueden suceder varios casos:

   - Los 100 ejemplos que comprueba pasan el test. En este caso veremos un mensaje del tipo:

     `OK, passed 100 tests`.

- Alguno de los ejemplos que genera aleatoriamente no ha pasado el test. Entonces, Haskell nos devuelve un contraejemplo.
- En el caso de las funciones que se cumplen bajo ciertas condiciones, puede ocurrir que de los 100 ejemplos aleatorios que ha creado Haskell, sólo unos pocos de ellos cumplan dicha condición. Será solamente esos pocos ejemplos que la cumplen, los que utilizará QuickCheck para comprobar dicha propiedad. Si esos ejemplos pasan el test, leeremos un mensaje parecido a lo siguiente:

  `Gave up! Passed only 48 tests.`

  En ese caso, es nuestra responsabilidad decidir si es suficiente o si debemos refinar algo más la propiedad.

Cabe mencionar el hecho de que si definimos propiedades en las que los tipos hayan sido creados por nosotros mismos, tenemos que proporcionarle a Haskell un *generador* específico de dicho tipo para que pueda generar los ejemplos aleatorios.

Además de la herramienta QuickCheck, Haskell nos permite usar Tipos Abstractos de Datos.

Para definir los Tipos Abstractos de Datos, introduzcamos primero la definición de abstracción.

La **abstracción** es un mecanismo para comprender problemas que involucran una gran cantidad de detalles. Algunos aspectos de la abstracción son *destacar* los detalles relevantes y *ocultar* los detalles irrelevantes.

Un **Tipo Abstracto de Datos (TAD)** es una colección de valores y operaciones que se definen mediante una especificación que es independiente de cualquier representación.

Por tanto, un TAD es una abstracción; se destacan los detalles (normalmente pocos) de la especificación (el qué), y se ocultan los detalles (normalmente numerosos) de la implementación (el cómo).

Haskell admite implementaciones del TAD; es decir, podemos usar distintas representaciones. Además, las funciones del código que usan contextos se han de definir de forma independientes de la representación siendo posible cambiar la representación sin necesidad de modificar dichas funciones.

Los **módulos** proporcionan la manera de construir tipos abstractos de datos en Haskell. Son estos módulos los que nos permiten exportar o importar la representación del TAD.

Cada representación consta de un módulo con las siguientes entradas:

- Nombre del módulo.
- Entre paréntesis aparecerán los nombres de las funciones que podremos usar en nuestro código principal. Son estas funciones las que se importan al utilizar TAD.
- A continuación aparecen definidas cada una de las funciones del punto anterior.

Veamos cómo funcionan los TADs.

1º) Supongamos que disponemos de la teoría matemática que queramos implementar en Haskell; es decir, tenemos un conjunto de operaciones cumpliendo varias propiedades.

2º) Implementación en Haskell. Elegimos una o varias representaciones de nuestra teoría en cuestión y utilizamos módulos para definir las funciones y las propiedades de las funciones que hemos nombrado en el apartado anterior.

3º) Importaremos una de las representaciones elegidas en el punto anterior y construiremos nuestro código principal que estará formado por funciones de nuestra teoría en estudio, las cuales serán independientes de la representación elegida; es decir, una vez formuladas en Haskell estas funciones, podremos modificar la representación sin necesidad de tener que volver a nuestro código principal y cambiar estas funciones.

Ejemplo 1 (TAD de las pilas)

1º) Una **pila** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que las operaciones de inserción y extracción se realizan por el mismo extremo. Las pilas también se llaman estructuras LIFO (del inglés Last In First Out), debido a que el último elemento en entrar será el primero en salir. Pensemos en una pila como una pila de platos.

El conjunto de operaciones de las pilas es el siguiente:

vacia: es la pila vacía.

apila x p: es la pila obtenida añadiendo x al principio de p.

cima p: es la cima de la pila p.

desapila p: es la pila obtenida suprimiendo la cima de p.

esVacia p: se verifica si p es la pila vacía.

El conjunto de propiedades de las pilas son las siguientes:

1. La cima de apilar x en una pila es x.
   `cima (apila x p) == x`
2. Si apilamos x en la pila p y luego desapilamos esa pila, obtenemos p.
   `desapila (apila x p) == p`
3. La pila vacía cumple que "es vacía".
   `esVacia vacia`
4. La pila que obtenemos al apilar x en una pila (ya sea ésta vacía o no) no es una pila vacía.
   `not (esVacia (apila x p))`

2º) Implementamos el TAD de las pilas. Para ello escogemos dos representaciones: las pilas como tipos de datos algebraicos y las pilas como listas.

1. Las pilas como tipos de datos algebraicos.

```
module PilaConTipoDeDatoAlgebraico
    (Pila,
     vacia,
     apila,
     cima,
     desapila,
     esVacia
    ) where

-- Tipo de dato algebraico de las pilas:
data Pila a = Vacia | P a (Pila a)
              deriving Eq
```

```
-- Procedimiento de escritura de pilas:
instance (Show a) => Show (Pila a) where
    showsPrec p Vacia cad = showChar '-' cad
    showsPrec p (P x s) cad =
        shows x (showChar '|' (shows s cad))

vacia :: Pila a
vacia = Vacia

apila :: a -> Pila a -> Pila a
apila x p = P x p

cima :: Pila a -> a
cima Vacia   = error "cima: pila vacia"
cima (P x _) = x

desapila :: Pila a -> Pila a
desapila Vacia   = error "desapila: pila vacia"
desapila (P _ p) = p

esVacia :: Pila a -> Bool
esVacia Vacia = True
esVacia _     = False

-- Ejemplo de pila:
p1 :: Pila Int
p1 = apila 1 (apila 2 (apila 3 vacia))

-- p1
-- 1|2|3|-

-- vacia
-- -

-- apila 4 p1
-- 4|1|2|3|-
```

```
-- cima p1
-- 1

-- desapila p1
-- 2|3|-

-- esVacia p1
-- False

-- esVacia vacia
-- True
```

2. Las pilas como listas.

```
module PilaConListas
    (Pila,
     vacia,
     apila,
     cima,
     desapila,
     esVacia,
    ) where

-- Tipo de dato de las pilas con listas:
newtype Pila a = P [a]
    deriving Eq

-- Procedimiento de escritura de pilas:
instance (Show a) => Show (Pila a) where
    showsPrec p (P [])     cad = showChar '-' cad
    showsPrec p (P (x:xs)) cad
        = shows x (showChar '|' (shows (P xs) cad))

vacia :: Pila a
vacia = P []

apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x:xs)
```

```
cima :: Pila a -> a
cima (P (x:_)) = x
cima (P [])    = error "cima de la pila vacia"

desapila :: Pila a -> Pila a
desapila (P [])     = error "desapila la pila vacia"
desapila (P (_:xs)) = P xs

esVacia :: Pila a -> Bool
esVacia (P xs) = null xs

-- Ejemplo de pila:
p1 :: Pila Int
p1 = apila 1 (apila 2 (apila 3 vacia))

-- p1
-- 1|2|3|-

-- vacia
-- -

-- apila 4 p1
-- 4|1|2|3|-

-- cima p1
-- 1

-- desapila p1
-- 2|3|-

-- esVacia p1
-- False

-- esVacia vacia
-- True
```

3º) Por último creamos un código de propiedades de las pilas en el que las funciones son independientes de la representación. En este código

importamos un módulo del punto anterior y comprobamos con QuickCheck que se cumplen dichas propiedades.

```
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

import PilaConTipoDeDatoAlgebraico
-- import PilaConListas

import Test.QuickCheck

-- Construimos un generador de pilas:
genPila :: (Num a, Arbitrary a) => Gen (Pila a)
genPila = do xs <- listOf arbitrary
             return (foldr apila vacia xs)

instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
    arbitrary = genPila

-- La cima de la pila que resulta de añadir x a la pila p
-- es x.
prop_cima_apila :: Int -> Pila Int -> Bool
prop_cima_apila x p = cima (apila x p) == x

-- quickCheck prop_cima_apila
-- OK, passed 100 tests.


-- La pila que resulta de desapilar después de añadir
-- cualquier elemento a una pila p es p.
prop_desapila_apila :: Int -> Pila Int -> Bool
prop_desapila_apila x p = desapila (apila x p) == p

-- quickCheck prop_desapila_apila
-- OK, passed 100 tests.


-- La pila vacía está vacía.
prop_vacia_esta_vacia :: Bool
prop_vacia_esta_vacia = esVacia vacia
```

```
-- quickCheck prop_vacia_esta_vacia
-- OK, passed 1 tests.


-- La pila que resulta de añadir un elemento en una pila
-- cualquiera no es vacía.
prop_apila_no_es_vacia :: Int -> Pila Int -> Bool
prop_apila_no_es_vacia x p = not (esVacia (apila x p))

-- quickCheck prop_apila_no_es_vacia
-- OK, passed 100 tests.
```

Capítulo 3

El retículo de los conceptos de un contexto formal

El AFC pretende representar retículos completos de objetos y sus propiedades extraídas de contextos formales, que son tablas de datos que representan relaciones binarias entre objetos y atributos (propiedades de los objetos). En esta teoría, un concepto formal es un par constituido por un conjunto de objetos (extensión) y un conjunto de atributos (intensión) de modo que la extensión está formada por todos los objetos que comparten los atributos dados, y la intensión la forman todos los atributos compartidos por los objetos dados. El conjunto de los conceptos de un contexto formal tiene estructura de retículo completo, lo que permite representarlos gráficamente como jerarquías conceptuales, posibilitando el análisis de estructuras complejas y descubriendo dependencias entre los datos.

Pasamos ahora a describir con detalle la teoría y su representación en Haskell.

3.1. Contextos formales

Definición 1 Un **contexto formal** $\mathbb{C} := (O, A, R)$ consta de dos conjuntos O y A y una relación R entre dichos conjuntos. Los elementos de O se llaman **objetos** del contexto y los de A, **atributos** del contexto. La relación R se denomina **relación de incidencia** del contexto.

Podemos representar un contexto por una tabla cruzada; es decir, por una tabla rectangular en la que las filas están encabezadas por los nombres

de los objetos y las columnas por los nombres de los atributos. Una cruz en una celda (o, a) significa que el objeto o tiene el atributo a.

Ejemplo 2
Consideremos el siguiente contexto sobre seres vivos:

	Necesita Agua	Acuático	Movilidad	Patas
Gato	X		X	X
Sanguijuela	X	X	X	
Rana	X	X	X	X
Maíz	X			
Pez	X	X	X	

Así por ejemplo podemos decir que el "gato" "necesita agua" y tiene "patas" y la "sanguijuela" también "necesita agua" pero no "tiene patas".

3.1.1. Representación del contexto formal como TAD

Para representar nuestro contexto formal como TAD hemos elegido cuatro representaciones distintas: con listas, con funciones, con matrices y con diccionarios. Tenemos varias ventajas de trabajar con TAD. Una de ellas es que estas cuatro representaciones son totalmente independientes entre sí. Además, el código principal compila y funciona correctamente con cualquiera de ellas sin necesidad de cambiar ninguna definición del código. Otra ventaja importante es que podemos completar el código de esta teoría en Haskell añadiendo nuevas representaciones; por ejemplo, usando conjuntos.

Las definiciones de cada módulo para construir el TAD son las siguientes:

Contexto: definiremos el tipo contexto. Cada una de las representaciones dependerá de este tipo.

creaContexto: recibe una lista de la forma `[(x,ys)]`, donde `ys` es la lista de atributos que posee el objeto `x`, y devuelve un contexto según la representación elegida.

objetos: recibe un contexto y devuelve la lista de los objetos del contexto.

atributos: recibe un contexto y devuelve la lista de los atributos del contexto.

relaciones: recibe un contexto y devuelve la lista de pares `[(x,y)]` donde `x` e `y` están relacionados, en el contexto.

Veamos cada una de las representaciones por separado:

TAD con listas

```
module ContextoConListas
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.List

-- Representación elegida:
data Contexto a b = C [a] [b] [(a,[b])]
                    deriving (Eq, Show)


creaContexto :: (Eq a, Eq b) => [(a,[b])] -> Contexto a b
creaContexto ps = C as bs ps
    where as = map fst ps
          bs = nub $ concatMap snd ps

-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C _ _ rs) = concatMap f rs
    where f (x,ys) = [(x,y) | y <-ys]
```

Como podemos observar nuestra primera representación se llama `ContextoConListas`.

Importamos el paquete `Data.List` puesto que vamos a trabajar con listas y usaremos funciones predefinidas en dicha librería.

Definimos el tipo `Contexto a b` mediante `C [a] [b] [(a,[b])]`. Es decir, un contexto de la forma `C xs ys ps` donde:

1. `xs` es la lista de los objetos del contexto.
2. `ys` es la lista de los atributos del contexto.
3. `ps` es una lista de pares en la que los primeros elementos de dichos pares son de tipo `a` y los segundos elementos de los pares son listas de tipo `b`; es decir, será una lista de pares en los que los primeros elementos serán objetos y su pareja del par será la lista de atributos relacionados con dicho objeto.

Ejemplo 3
Veamos en nuestro código principal como vendría representado el ejemplo 2.

```
-- ejd
-- C "gsrmp" [1,3,4,2] [('g',[1,3,4]),('s',[1,2,3]),
--           ('r',[1,2,3,4]),('m',[1]),('p',[1,2,3])]
```

TAD con funciones

```
module ContextoConFunciones
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.List

-- Representación elegida:
data Contexto a b = C [a] [b] (a -> b -> Bool)
```

```
creaContexto :: (Eq a, Eq b) => [(a,[b])] -> Contexto a b
creaContexto ps = C as bs f
    where as = map fst ps
          bs = nub $ concatMap snd ps
          f x y = x `elem` as && y `elem` img ps x

img ps x = head [ys | (x',ys) <- ps, x' == x]

-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C as bs f) = [(x,y) | x <- as, y <- bs, f x y]
```

Esta segunda representación se llama `contextoConFunciones`.

Al igual que antes, importamos `Data.List` porque usaremos algunas funciones predefinidas en esta librería.

En este caso, describimos el tipo contexto `Contexto a b` como `C xs ys f`, donde:

1. `xs` es una lista de elementos tipo `a`; que serán los objetos del contexto.
2. `ys` es una lista de elementos tipo `b`; que serán los atributos del contexto.
3. `f` es una función que toma como argumentos de entrada dos elementos; uno de tipo `a` y otro de tipo `b` y devuelve un booleano que dependerá de si el objeto `a` está relacionado con el atributo `b` o no.

TAD con matrices

```
module ContextoConMatrices
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.Matrix
import Data.List

-- Representación elegida:
data Contexto a b = C [a] [b] (Matrix Int)
                    deriving (Eq, Show)


creaContexto :: (Eq a, Eq b, Ord b) =>
                [(a,[b])] -> Contexto a b
creaContexto ps = C as bs m
     where as = map fst ps
           bs = sort $ nub $ concatMap snd ps
           m = deListaParesAmatriz ps

deListaParesAmatriz :: (Eq a, Eq b, Ord b) =>
                       [(a,[b])] -> Matrix Int
deListaParesAmatriz ps = matrix m n f
    where as = map fst ps
          m = length as
          bs = sort $ nub $ concatMap snd ps
          n = length bs
          relacionado ps x y = x `elem` as &&
                               y `elem` img ps x
          f (i,j)| relacionado ps (as !! (i-1)) (bs !! (j-1))
                                  = 1
                 | otherwise      = 0

img ps x = head [ys | (x',ys) <- ps, x' == x]
```

```
-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C as bs m) = [(x,y) | x <- as, y <- bs,
                          getElem (posicion x as)
                                  (posicion y bs) m == 1]

posicion x xs = head [y | (x',y) <- zip xs [1..], x == x']
```

Llamamos a esta nueva representación `codigoConMatrices`. A continuación importamos tanto la librería `Data.Matrix` como la de `Data.List` pues, a lo largo del código, usaremos algunas de sus funciones predefinidas.

El tipo `Contexto a b` en este caso, es de la forma `C xs ys m` donde:

1. `xs` es una lista de elementos de tipo `a`; los objetos del contexto.
2. `ys` es una lista de elementos de tipo `b`; los atributos del contexto.
3. `m` es una matriz en la que cada celda *(i,j)* será 0 ó 1. Si hay un 1 representará que el objeto de la fila i posee el atributo j; es decir, el par `(i,j)` estará en las relaciones del contexto y si, por el contrario hay un 0, significará que ese objeto no tiene dicho atributo (el par `(i,j)` no estará en el conjunto de las relaciones).

Comentemos el `creaContexto` de esta representación.

Como en las demás ocasiones, la función `creaContexto` recibe una lista de pares y devuelve un contexto, según la representación anterior. Es decir, tomando como argumento de entrada una lista de pares en los que los primeros elementos de cada par son del tipo `a` y los segundos elementos de cada par son listas de elementos del tipo `b`, obtendríamos un contexto `C as bs m` donde `as` y `bs` son las listas de los objetos y la de los atributos, respectivamente. Veamos cómo obtenemos la matriz `m`.

La matriz `m` está definida como `deListaParesAmatriz ps` donde `deListaParesAmatriz ps` es una matriz de dimensiones `m` (longitud de la

lista de los objetos) y `n` (longitud de la lista de los atributos) y las entradas de dicha matriz vienen dadas por los valores de la función `f`.

Esta función `f` toma como argumento de entrada un par `(i,j)` que representará una celda de la matriz y nos devuelve un 1 en la posición *(i,j)* si el elemento del conjunto de los objetos `as` de la posición *i-1* (escojemos la posición *i-1* en vez de la posición *i* porque en Haskell las listas se empiezan a contar desde la posición 0) está relacionado con el elemento de la posición *j-1* de la lista de los atributos `bs`. La propiedad "estar relacionado" será cierta si la función `relacionado ps x y` lo es; para `ps` la lista de pares que le dimos como argumento de entrada a nuestra función principal `creaContexto`.

Ahora bien, la función `relacionado` devuelve `True` si el elemento `x` pertenece a la lista de los objetos (`as`) y el elemento `y` pertenece a lo que hemos llamado `img ps x`; es decir, recorremos la lista de pares `ps` y nos quedamos con el segundo elemento del par, `ys`, cuyo primer elemento es `x`.

En definitiva, la matriz `m` será una matriz de dimensiones (número de objetos) $\times$ (número de atributos), en la que las entradas *(i,j)* serán 1 ó 0 dependiendo de si el objeto i tiene el atributo j o no.

Hay que mencionar el hecho de que los auxiliares definidos para esta función no se importan en el código principal; es decir, si queremos usar en nuestro código principal las funciones `deListaParesAmatriz` o `img` debemos definirlas de nuevo ya que sólo se importarán las 5 funciones de acceso de las representaciones.

Ejemplo 4
Observemos la representación del ejemplo 2 con las matrices:

```
-- ejd
-- C "gsrmp" [1,2,3,4] ( 1 0 1 1 )
--                     ( 1 1 1 0 )
--                     ( 1 1 1 1 )
--                     ( 1 0 0 0 )
--                     ( 1 1 1 0 )
```

TAD con diccionarios

```
module ContextoConDiccionarios
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.List
import Data.Ix
import qualified Data.Map as M

-- Representación elegida:
data Contexto a b = C [a] [b] (M.Map a [b])
                    deriving (Eq, Show)


creaContexto :: (Ord a, Eq b) => [(a,[b])] -> Contexto a b
creaContexto ps = C as bs (foldr f M.empty ps)
    where as = map fst ps
          bs = nub $ concatMap snd ps
          f (x,ys) d = M.insert x ys d

-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C _ _ ds) = concatMap f (M.toList ds)
    where f (x,ys) = [(x,y) | y <- ys]
```

Denominamos a esta última representación `contextoConDiccionarios`. En este caso, el tipo `Contexto a b` viene dado por `C xs ys d` donde:

1. `xs` es una lista de elementos de tipo `a` que será la lista de los objetos del contexto.

2. `ys` es una lista de elementos de tipo `b` que será la lista de los atributos del contexto.

3. `d` es un diccionario cuyas claves son los objetos del contexto y los valores asociados son los atributos de cada objeto.

Ejemplo 5
El ejemplo 2, utilizando la representación con diccionarios, vendrá dado por:

```
-- ejd
-- C "gsrmp" [1,3,4,2]
-- (fromList [('g',[1,3,4]),('m',[1]),('p',[1,2,3]),
-- ('r',[1,2,3,4]),('s',[1,2,3])])
```

Una vez que tenemos en Haskell las cuatro representaciones del contexto formal como TAD, podemos continuar con el cógido principal. Para dicho código necesitamos definir un generador de contextos ya que hemos creado un tipo nuevo; el tipo `Contexto a b`. Este generador es el siguiente:

```
transforma:: (Eq a, Eq b) => [(a,b)] -> [(a,[b])]
transforma rs = [(x,img x) | x <- as]
    where as = nub $ map fst rs
          img x = [y | (x', y) <- rs, x' == x]


genContexto :: Gen (Contexto Int Int)
genContexto = do
  xs <- listOf1 arbitrary
  let xs1 = nub $ map abs xs
  ys <- listOf (elements [(x,y) | x <- xs1, y <- xs1])
  return (creaContexto (transforma ys))

instance Arbitrary (Contexto Int Int) where
    arbitrary = genContexto
```

Veamos cómo funciona este generador. La función principal la hemos llamado `genContexto`. Para ella, necesitamos otra función auxiliar llamada `transforma rs` donde `rs` será una lista de pares de elementos.

La función `genContexto` hace lo siguiente:

1. Crea una lista no vacía `xs` de longitud arbitraria.

2. Nombra como `xs1` a la lista de los elementos sin repetición de `xs` y toma cada uno de ellos en valor absoluto.

3. Crea una lista `ys` de longitud aleatoria formada por pares de elementos `(x,y)` tales que `x` e `y` provienen de la lista `xs1` del paso 2.

4. Por último devuelve un `creaContexto (transforma ys)` donde la función `transforma ys` crea una lista de pares en los que los primeros elementos vienen de la lista `as` (que será la lista de todos los primeros elementos de `ys` sin repetición) y los segundos elementos del par serán `img x`, donde `img x` será la lista de los elementos `y` tales que el par `(x,y)` aparecía en la lista `ys`. Es decir, `transforma rs` toma como argumento de entrada una lista de pares de elementos y devuelve otra lista de pares pero en los que los segundos elementos de cada par son listas.

Tengamos en cuenta que dependiendo del TAD que importemos en nuestro código en cada momento, esta función generará contextos de una manera o de otra; dependiendo de la representación usada en dicho TAD.

Además, este generador devuelve contextos aleatorios en los que los elementos de los conjuntos de objetos y de atributos son números enteros. Luego, es importante saber que, para cualquier propiedad que queramos comprobar con *quickCheck* debemos usar el tipo `Int`.

3.1.2. Propiedades del contexto formal

Una vez tenemos definido el contexto formal, estamos en condiciones de estudiar ciertas propiedades. Por ejemplo, podemos obtener los atributos que estén relacionados con un objeto particular y los objetos que se relacionen con un atributo; es decir, aquellos objetos que tienen una determinada propiedad.

Dado un objeto $o \in O$, el conjunto de atributos relacionados con él viene dado por:

$$o' = \{a \in A \,|\, oRa\}$$

Debido a que Haskell permite la definición de listas por comprensión, podemos definir este conjunto como los elementos que provengan de la lista de los atributos del contexto que cumplan la condición oRa.

```
atributosElto :: (Eq a, Eq b) => Contexto a b -> a -> [b]
atributosElto c o =
        [a | a <- atributos c, (o,a) `elem` relaciones c]

-- atributosElto ejd 'g'
-- [1,3,4]
-- atributosElto ejd 's'
-- [1,3,2]
```

De manera análoga, dado un atributo $a \in A$, el conjunto de objetos relacionados con él viene dado por:

$$a' = \{o \in O \,|\, oRa\}$$

Y, de la misma manera; mediante listas por comprensión, definimos este conjunto en Haskell:

```
objetosElto :: (Eq a, Eq b) => Contexto a b -> b -> [a]
objetosElto c a =
            [o | o <- objetos c, (o,a) `elem` relaciones c]

-- objetosElto ejd 1
-- "gsrmp"
-- objetosElto ejd 4
-- "gr"
```

En ambas definiciones se ha utilizado la función predefinida `elem x ys` (también puede escribirse como `x `elem` ys`), que se verifica si `x` pertenece a `ys`.

Ahora bien, con el fin de expresar que un objeto o está relacionado, mediante la relación R, con un atributo a, escribiremos oRa o bien $(o, a) \in R$ y lo leeremos del siguiente modo: "el objeto o **tiene** el atributo a".

Esta definición en Haskell es de lo más natural posible pues dos elementos están relacionados si su par corresponiente pertenece al conjunto de las relaciones del contexto.

```
relacionado:: (Eq a, Eq b) => Contexto a b -> a -> b -> Bool
relacionado c o a = (o,a) `elem` relaciones c

-- El objeto "gato" tiene la propiedad "tener patas" pero no
-- tiene la de "ser acuático".
-- relacionado ejd 'g' 4
-- True
-- relacionado ejd 'g' 2
-- False
```

Debido a que a lo largo del texto será útil trabajar con subconjuntos de objetos y de atributos, definamos por un lado el conjunto potencia de los objetos y por otro, el conjunto potencia de los atributos. Una vez lo tengamos, será fácil tomar subconjuntos que provengan de ellos.

Definición 2 Dado un contexto (O, A, R), se define el **conjunto potencia del conjunto de objetos** como el conjunto de todos los subconjuntos de O.

Para la definición en Haskell de esta función vamos a utilizar la función predefinida `subsequences` que viene dada, justamente, por el conjunto de sublistas del argumento que le demos de entrada.

```
obss :: (Eq a, Eq b) => Contexto a b -> [[a]]
obss = subsequences . objetos
```

De forma similar, podemos definir el **conjunto potencia del conjunto de atributos** de dicho contexto.

```
atbs :: (Eq a, Eq b) => Contexto a b -> [[b]]
atbs = subsequences . atributos
```

Cabe destacar el hecho de que podemos utilizar la composición de funciones porque las funciones en Haskell pueden ser de **orden superior**[1].

[1]Una función es de **orden superior** si toma una función como argumento o devuelve una función como resultado.

Hemos visto el conjunto de atributos relacionados con un único objeto (y el conjunto de objetos que poseen un único atributo). Es el momento de pararnos a pensar qué ocurre si, en vez de estudiar el hecho para un único objeto (o atributo), lo hacemos para un conjunto de ellos.

Definición 3 Dado un conjunto de objetos $G \subseteq O$ definimos la **intensión** de G;

$$G' := \{a \in A \mid oRa \text{ para todo } o \in G\},$$

como el conjunto de atributos comunes a todos los objetos de G.

De nuevo nos encontramos con conjuntos que se pueden definir fácilmente en Haskell mediante listas por comprensión.

```
intension:: (Eq a, Eq b) => Contexto a b -> [a] -> [b]
intension c gs =
    [a | a <- atributos c, all (\g -> relacionado c g a ) gs]

-- intension ejd "gs"
-- [1,3]
```

De manera análoga, para un conjunto M de atributos se define su **extensión**;

$$M' := \{o \in O \mid oRa \text{ para todo } a \in M\},$$

como el conjunto de objetos que tienen todos los atributos de M.

```
extension:: (Eq a, Eq b) => Contexto a b -> [b] -> [a]
extension c ms =
    [o | o <- objetos c, all (\m -> relacionado c o m) ms]

-- extension ejd [2,4]
-- "r"
```

En este caso, la propiedad que deben cumplir los elementos que incluiremos en la lista que estamos formando usa la función predefinida `all p xs` que se verifica si todos los elementos de `xs` cumplen la propiedad `p`.

La intensión y la extensión cumplen las siguientes propiedades:

Proposición 1
La intensión del conjunto vacío es el conjunto de atributos del contexto.

```
intension_vacio:: Contexto Int Int -> Bool
intension_vacio c = intension c [] == atributos c

-- quickCheck intension_vacio
-- OK, passed 100 tests.
```

La extensión del conjunto vacío es el conjunto de objetos del contexto.

```
extension_vacio:: Contexto Int Int -> Bool
extension_vacio c = extension c [] == objetos c

-- quickCheck extension_vacio
-- OK, passed 100 tests.
```

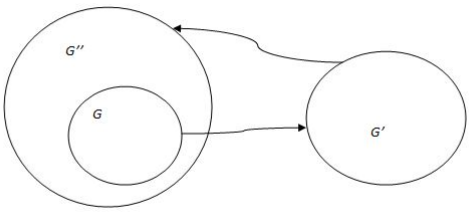
La *demostración* de esta proposición es trivial pues se obtiene directamente de las definiciones de intensión y extensión.

A continuación definimos los operadores de clausura.

Definición 4 Dado un conjunto de objetos G, la **clausura** de G con respecto al contexto (O, A, R), representada por G'', es el conjunto de objetos del contexto que poseen todos los atributos de G'.

Definimos la clausura en Haskell mediante la composición de funciones ya que dado un subconjunto G del conjunto de los objetos, primero hacemos su intensión; G', y después su extensión $(G')' = G''$.

```
clausuraO :: (Eq a, Eq b) => Contexto a b -> [a] -> [a]
clausuraO c = extension c . intension c
```



Para un conjunto de atributos M, su **clausura** con respecto al contexto (O, A, R), denotada por M'', será el conjunto de atributos del contexto que poseen todos los objetos de M'.

```
clausuraA :: (Eq a, Eq b) => Contexto a b -> [b] -> [b]
clausuraA c = intension c . extension c
```

Las siguientes proposiciones describen ciertas propiedades de los contextos formales.

Proposición 2
Sea (O, A, R) un contexto y sean $G, G_1, G_2 \subseteq O$ conjuntos de objetos, entonces se cumplen las siguientes propiedades:

1. $G \subseteq G''$
2. $G_1 \subseteq G_2 \Rightarrow G_2{}' \subseteq G_1{}'$
3. $G' = G'''$

Para introducir esta propiedad, tengamos en cuenta que dados dos conjuntos cualesquiera X e Y, se dirá que X es subconjunto de Y si todos los elementos de X están contenidos en el conjunto Y.

Escribimos esta definición en Haskell comprobando, mediante una lista por comprensión, si todos los elementos del primer conjunto están incluidos en el segundo. Para ello utilizamos la función predefinida `and xs` que devolverá `True` si todas las entradas de la lista `xs` son `True`.

```
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]
```

Veamos ahora el primer apartado:

```
prop_2_1:: Contexto Int Int -> [Int] -> Property
prop_2_1 c gs =
    (not (null obs) && subconjunto gs obs) ==>
     subconjunto gs (clausuraO c gs)
      where obs = objetos c

-- quickCheck prop_2_1
-- *** Gave up! Passed only 48 tests.
```

De los 100 ejemplos aleatorios, sólo se cumple la propiedad para 48 de ellos (pues sólo 48 de esos test cumplen la condición para que la propiedad pueda ser probada), pero podemos mejorarla ya que *QuickCheck* escoge contextos y conjuntos de manera aleatoria y no se le está exigiendo que dichos conjuntos estén contenidos en el conjunto de objetos de los contextos obtenidos de forma aleatoria. Luego, impongámosle que G sea un subconjunto del conjunto objeto.

```
prop_2_1':: Contexto Int Int -> Bool
prop_2_1' c =
    and [subconjunto gs (clausura0 c gs) | gs <- obss c]

-- quickCheckWith (stdArgs {maxSize=6}) prop_2_1'
-- +++ OK, passed 100 tests.
```

Apartado segundo:

De la misma manera que en el apartado anterior, impongámosle de entrada que el conjunto G_1 sea un subconjunto del conjunto de los objetos y G_2 sea otro subconjunto del conjunto de los objetos que tenga incluído al conjunto G_1.

```
prop_2_2 :: Contexto Int Int -> Bool
prop_2_2 c =
    and [subconjunto (intension c gs2) (intension c gs1)|
                          gs1 <- obss c,
                          gs2 <- extiendeSub gs1 obs]
        where obs = objetos c

extiendeSub ys xs = map (ys++) $ subsequences (xs \\ ys)

-- quickCheckWith (stdArgs {maxSize= 22}) prop_2_2
-- +++ OK, passed 100 tests.
```

Podemos observar que la lista creada por comprensión tiene dos generadores; `gs1` y `gs2`, y además, el segundo generador depende del primero. Es decir, usamos lo que se denomina en Haskell **generadores dependientes**.

Tercer apartado:

En Haskell definiremos este tercer apartado de la siguiente manera: para todo subconjunto del conjunto de los objetos del contexto, se verifica la propiedad.

```
prop_2_3 :: Contexto Int Int -> Bool
prop_2_3 c =
    and [intension c gs == intension c (clausura0 c gs)|
              gs <- obss c]

-- quickCheckWith (stdArgs {maxSize= 22}) prop_2_3
-- +++ OK, passed 100 tests.
```

DEMOSTRACIÓN:

1. Si $o \in G$, entonces oRa para todo $a \in G'$ lo que implica que $o \in G''$.

2. Si $a \in G_2{}'$, entonces oRa para todo $o \in G_2$. En particular oRa para todo $o \in G_1$ ya que $G_1 \subseteq G_2$ y por tanto $a \in G_1{}'$.

3. Por (1) tenemos que $G' \subseteq G'''$, y de $G \subseteq G''$ obtenemos $G''' \subseteq G'$ por (2). Luego, tenemos la igualdad.

□

Proposición 3

Sea (O, A, R) un contexto y sean $M, M_1, M_2 \subseteq A$ conjuntos de atributos, entonces se cumplen las siguientes propiedades:

1. $M \subseteq M''$

2. $M_1 \subseteq M_2 \Rightarrow M_2' \subseteq M_1'$

3. $M' = M'''$

Primer apartado:

Definiremos este apartado en Haskell como: para todo subconjunto M del conjunto de los atributos del contexto, se cumple $M \subseteq M''$.

```
prop_3_1:: Contexto Int Int -> Bool
prop_3_1 c =
    and [subconjunto ms (clausuraA c ms) | ms <- atbs c]

-- quickCheckWith (stdArgs {maxSize= 6}) prop_3_1
-- +++ OK, passed 100 tests.
```

Apartado segundo:

De manera análoga a la proposición anterior, definimos este apartado como una lista por comprensión en la que los dos generadores son dependientes.

```
prop_3_2 :: Contexto Int Int -> Bool
prop_3_2 c =
    and [subconjunto (extension c ms2) (extension c ms1)|
                          ms1 <- atbs c,
                          ms2 <- extiendeSub ms1 atb]
        where atb = atributos c

-- quickCheckWith (stdArgs {maxSize= 22}) prop_3_2
-- +++ OK, passed 100 tests.
```

Tercer apartado:

Para todo subconjunto del conjunto de atributos del contexto se verifica $M' = M'''$.

```
prop_3_3 :: Contexto Int Int -> Bool
prop_3_3 c =
    and [extension c ms == extension c (clausuraA c ms)|
          ms <- atbs c]

-- quickCheckWith (stdArgs {maxSize= 6}) prop_3_3
-- +++ OK, passed 100 tests.
```

DEMOSTRACIÓN:

1. Si $a \in M$, entonces oRa para todo $o \in M'$ lo que implica que $a \in M''$.

2. Si $o \in M_2'$, entonces oRa para todo $a \in M_2$. En particular oRa para todo $a \in M_1$ ya que $M_1 \subseteq M_2$ y por tanto $o \in M_1'$.

3. Por (1) tenemos que $M' \subseteq M'''$, y de $M \subseteq M''$ obtenemos $M''' \subseteq M'$ por (2). Luego, tenemos la igualdad.

□

Proposición 4
Sea (O, A, R) un contexto, $G \subseteq O$ un conjunto de objetos y $M \subseteq A$ un conjunto de atributos, entonces:

$$G \subseteq M' \iff M \subseteq G'$$

Veamos la implicación de izquierda a derecha:
$G \subseteq M' \Longrightarrow M \subseteq G'$
Reescribimos la propiedad de la siguiente forma:
$\forall M \in \mathcal{P}(A)$ y $\forall G \in \mathcal{P}(M')$ se verifica que $M \subseteq G'$
para adecuarla a su definición en Haskell.

```
prop_4_dcha :: Contexto Int Int -> Bool
prop_4_dcha c =
    and [subconjunto ms (intension c gs) | ms <- atbs c,
          gs <- subsequences (extension c ms)]

-- quickCheckWith (stdArgs {maxSize= 20}) prop_4_dcha
-- +++ OK, passed 100 tests.
```

Pasemos a la otra implicación:
$G \subseteq M' \Longleftarrow M \subseteq G'$
Definimos esta implicación en Haskell como:
$\forall G \in \mathcal{P}(O)$ y $\forall M \in \mathcal{P}(G')$ se verifica que $G \subseteq M'$.

```
prop_4_izda :: Contexto Int Int -> Bool
prop_4_izda c =
    and [subconjunto gs (extension c ms) | gs <- obss c,
          ms <- subsequences (intension c gs)]

-- quickCheckWith (stdArgs {maxSize= 20}) prop_4_izda
-- +++ OK, passed 100 tests.
```

Cabe mencionar que ambas definiciones en Haskell de las dos implicaciones utilizan generadores dependientes en la construcción de la lista por comprensión.

DEMOSTRACIÓN:

$\boxed{\Rightarrow}$ $a \in M$ ¿$a \in G'$?

Si $a \in M$ entonces $o \in M'$ con oRa. Como $G \subseteq M'$ tenemos que $o \in G$ implica oRa. Lo que nos lleva a que $a \in G'$ pues $oRa \, \forall o$.

$\boxed{\Leftarrow}$ $o \in G$ ¿$o \in M'$?

$a \in M$ por lo que $a \in G'$ pues $M \subseteq G'$. Si $a \in G'$, entonces $o \in G$ con oRa. Si $a \in M$ entonces oRa (pues todos los $a \in M$ están en G' y por tanto cumplen oRa), de donde obtenemos que $o \in M'$.

□

Proposición 5
Sea (O, A, R) un contexto y sean $G \subseteq O$ un conjunto de objetos y $M \subseteq A$ un conjunto de atributos, entonces:

$$M \subseteq G' \Longleftrightarrow G \times M \subseteq R$$

Antes de comprobar que se cumple esta doble implicación, introduzcamos la definición de producto cartesiano en Haskell.

Definición 5 Dados dos conjuntos X e Y cualesquiera, se define el **producto cartesiano** de X e Y como el conjunto de pares en el que el primer elemento de cada par es un elemento del conjunto X y el segundo elemento del par es un elemento del conjunto Y.

La definición en Haskell es la siguiente:

```
prodCartesiano:: [a] -> [b] -> [(a,b)]
prodCartesiano xs ys = [(x,y) | x <- xs, y <- ys]
```

Veamos ahora que se satisfacen las dos implicaciones:

$M \subseteq G' \Longrightarrow G \times M \subseteq R$

Para definir esta implicación en Haskell, la escribimos equivalentemente de la siguiente forma:

$\forall G \in \mathcal{P}(O)$ y $\forall M \in \mathcal{P}(G')$ se verifica que $G \times M \subseteq R$

Como vemos, M depende de G. Luego en la definición en Haskell, los generadores de la lista por comprensión serán dependientes.

```
prop_5_dcha :: Contexto Int Int -> Bool
prop_5_dcha c =
    and [subconjunto (prodCartesiano gs ms) rel |
         gs <- obss c, ms <- subsequences (intension c gs)]
        where rel = relaciones c

-- quickCheckWith (stdArgs {maxSize= 20}) prop_5_dcha
-- +++ OK, passed 100 tests.
```

$M \subseteq G' \Longleftarrow G \times M \subseteq R$

De manera análoga a lo comentado anterioremente, lo que realmente definiremos en este código es lo siguiente:

$\forall G \in \mathcal{P}(O)$, $\forall M \in \mathcal{P}(A)$ tales que $G \times M \subseteq R$ se verifica que $M \subseteq G'$

```
prop_5_izda :: Contexto Int Int -> Bool
prop_5_izda c =
    and [subconjunto ms (intension c gs) |
         gs <- obss c, ms <- atbs c,
         subconjunto (prodCartesiano gs ms) rel]
        where rel = relaciones c

-- quickCheckWith (stdArgs {maxSize= 20}) prop_5_izda
-- +++ OK, passed 100 tests.
```

La *demostración* de esta proposición se tiene directamente de la definición.

Pasemos a introducir una última propiedad de los contextos formales.

Proposición 6
Sea G una familia no vacía de conjuntos de objetos de un contexto formal. Se tiene que:

$$\left(\bigcup G\right)' = \bigcap \{X' : X \in G\}$$

Análogamente, esta proposición se cumple para una familia M no vacía de conjuntos de atributos de un contexto formal:

$$\left(\bigcup M\right)' = \bigcap \{Y' : Y \in M\}$$

Observemos que en esta proposición usamos la unión y la intersección generalizada de conjuntos luego, definámoslas.

Definición 6 La **unión generalizada** de una lista de listas es la lista formada por todos los elementos de cada lista.

```
unionG :: Eq a => [[a]] -> [a]
unionG = foldl union []
```

Para la definición en Haskell de esta función usamos la función predefinida `foldl union [] (x:xs)`. Esta función trabaja del siguiente modo: `foldl union [] (x:xs) = foldl union (union [] x) xs`; es decir, en el primer paso hace la union de `[]` y `x` y a continuación vuelve a hacer `foldl` pero en el lugar del valor inicial pondremos el resultado `union [] x` y así sucesivamente hasta que la lista `(x:xs)` sea vacía. Haskell tiene predefinida la función `union xs ys` que es justamente la unión de `xs` e `ys`.

Definición 7 La **intersección generalizada** de una lista de listas es la lista formada por los elementos comunes a todas las listas.

```
interseccionG :: Eq a => [[a]] -> [a]
interseccionG = foldl1 intersect
```

En esta representación estamos usando la función `foldl1 intersect (x:xs)`. Lo que hace esta función es similar a lo explicado anteriormente con el `foldl` pero paramos cuando la lista contenga un solo elemento. En esta definición utilizamos la función predefinida de Haskell `intersect xs ys` que es justamente la intersección de `xs` e `ys`; es decir, los elementos de `xs` que también pertenecen a `ys`.

Una vez que tenemos la definición de la unión y la intersección, estamos en condiciones de comprobar si se cumple la última proposición. Primero comprobaremos que se verifica para el caso de los objetos y a continuación para el caso de los atributos.

Haskell nos permite definir estas dos propiedades de la manera más natural posible. Aún así, debajo del código daremos una aclaración.

```
prop_6o :: Contexto Int Int -> Bool
prop_6o c =
     and [intension c (unionG gss) ==
          interseccionG [intension c xs| xs <- gss] |
            gss <- lss]
         where lss = tail $ subsequences (obss c)

-- quickCheckWith (stdArgs {maxSize= 8}) prop_6o
-- +++ OK, passed 100 tests.
```

Dado un contexto `c`$= (O, A, R)$ podemos obtener `obss c`$= (\mathcal{P}(O))$. Sea ahora `ls`$= \mathcal{P}(\mathcal{P}(O)) - \{\emptyset\}$, que representa las posibles familias no vacías de conjuntos de objetos del contexto `c`.

Mediante `gss <- lss` se recorren todas las familias de `lss`. Luego el `gss` del código Haskell se corresponde con el conjunto G del enunciado de la proposición 6. Por último, para cada `gss <- lss`, se comprueba que `(unionG gss)`$' =$ `interseccionG [xs' | xs <- gss]`.

De manera análoga al caso de los objetos, definimos la propiedad en Haskell para los atributos.

```
prop_6a :: Contexto Int Int -> Bool
prop_6a c =
     and [intension c (unionG mss) ==
          interseccionG [intension c ys| ys <- mss] |
            mss <- lss]
         where lss = tail $ subsequences (atbs c)

-- quickCheckWith (stdArgs {maxSize= 10}) prop_6a
-- +++ OK, passed 100 tests.
```

DEMOSTRACIÓN:

Hacemos la demostración para el caso de los objetos. Es análoga para el caso de los atributos.

$$\begin{aligned} a \in (\bigcup G)' &\iff oRa \text{ para todo } o \in \bigcup G \\ &\iff oRa \text{ para todo } o \in G \\ &\iff a \in X' \text{ donde } X' \in G \\ &\iff a \in \bigcap\{X' \mid X' \in G\} \quad \square \end{aligned}$$

3.2. Conceptos formales

La noción de concepto relativo a un contexto formal sintetiza la idea de un conjunto de objetos, caracterizados por el conjunto de atributos que poseen. Dado un contexto formal, $\mathcal{C}$, se considera que un concepto que se puede "extraer" de $\mathcal{C}$ está formado por un conjunto de objetos de $\mathcal{C}$, junto con el conjunto de atributos que los caracterizan.

3.2.1. Definiciones

La definición formal de la noción de concepto es la siguiente:

Definición 8 Un **concepto formal** de un contexto (O, A, R) es un par (G, M) con $G \subseteq O$, $M \subseteq A$, $G' = M$ y $M' = G$. Denotamos por G la **extensión** y M la **intensión** de dicho concepto.

```
esConcepto :: (Ord a, Ord b) =>
              Contexto a b -> ([a],[b]) -> Bool
esConcepto c (gs, ms) = intension c gs == ms &&
                        extension c ms == gs
```

Si comparamos la definición con la función correspondiente en Haskell nos fijamos en que faltan dos condiciones para comprobar si un par es concepto de un contexto. Estas condiciones serían $G \subseteq O$ y $M \subseteq A$. Sin embargo, no es necesario incluirlas pues si, por ejemplo $G \nsubseteq O$, se tendría $G' = [\,]$ y debería de cumplirse que $[\,] = M$. Pero por otro lado, tenemos por definición que $M' = [\,]' = O$ y como $G \nsubseteq O$ no puede ser $M' = G$. Se probaría de manera similar que no sería necesario incluírlas si fuera $M \nsubseteq A$.

Definición 9 El conjunto de conceptos de un contexto $\mathcal{C}$ viene dado por:

```
conceptos :: (Ord a, Ord b) => Contexto a b -> [([a],[b])]
conceptos c = [(gs,ms)| gs <- obss c,
               let ms = intension c gs,
               extension c ms == gs]

-- conceptos ejd
-- [("r",[1,3,4,2]),("gr",[1,3,4]),("srp",[1,3,2]),
```

```
-- ("gsrp",[1,3]),("gsrmp",[1])]
-- (0.27 secs, 6506296 bytes)
```

Observemos que con esta definición obtenemos la lista de todos los conceptos de un contexto mediante listas por comprensión; es decir, de la forma más natural posible. Tomamos la definición e imponemos que sólo escoja los subconjuntos del conjunto de objetos (`gs`) tales que su intensión (`ms`) coincide con `gs`. En este caso, formamos el concepto `(gs, ms)`. Se puede suponer que esta definición, aunque es correcta, no es la mejor pues tiene que recorrer todo el conjunto potencia del conjunto objeto (que tendrá longitud 2^n -siendo n el número de objetos-) y a medida que lo va recorriendo, va comprobando para cada uno de los conjuntos si cumple una propiedad; lo que podría llegar a ser inviable. Luego, esta forma no es muy buena desde el punto de vista computacional. Por ello, más adelante incluiremos algoritmos para la obtención de todos los conceptos de un contexto que serán más eficientes.
El número de conceptos de un contexto viene dado por:

```
numConceptos :: (Ord a, Ord b) => Contexto a b -> Int
numConceptos = length . conceptos

-- numConceptos ejd
-- 5
```

La siguiente función sirve para escribir todos los conceptos dispuestos uno debajo del otro, en vez de recogidos en una lista:

```
display :: Show a => [a] -> IO()
display [] = return ()
display (x:xs) = do print x
                    display xs
```

Nota 1 Habría que escribir `display (conceptos c)` donde `c` sería nuestro contexto $\mathcal{C}$.

3.2.2. Retículo de los conceptos

Una vez que tenemos la definición de concepto formal, podemos introducir la definición de subconcepto.

Definición 10 Si (G_1, M_1) y (G_2, M_2) son conceptos de un contexto (O, A, R), entonces se dirá que (G_1, M_1) es un **subconcepto** de (G_2, M_2), siempre que $G_1 \subseteq G_2$ (o lo que es lo mismo, $M_2 \subseteq M_1$). En este caso, (G_2, M_2) es un **superconcepto** de (G_1, M_1).

Denotaremos esta relación como $(G_1, M_1) \leq (G_2, M_2)$.

La definimos en Haskell comprobando que se verifican las propiedades para ser subconcepto.

```
esSubconcepto :: (Ord a, Ord b) =>
              Contexto a b -> ([a],[b]) -> ([a],[b]) -> Bool
esSubconcepto c (gs1,ms1) (gs2,ms2) = subconjunto gs1 gs2 &&
                                      esConcepto c (gs1,ms1) &&
                                      esConcepto c (gs2,ms2)
```

Existe una herramienta llamada **conexp** [2] que nos permite representar un contexto mediante una tabla cruzada tal y como vimos en el ejemplo 2 y a partir de aquí, obtener toda la información que este contexto nos proporciona. Además, podemos obtener el retículo de conceptos del contexto de un modo gráfico mediante un diagrama de líneas lo que nos permitirá observar con facilidad qué conceptos son subconceptos (o superconceptos) de otros.

Veamos cómo representamos el ejemplo 2 en este programa:

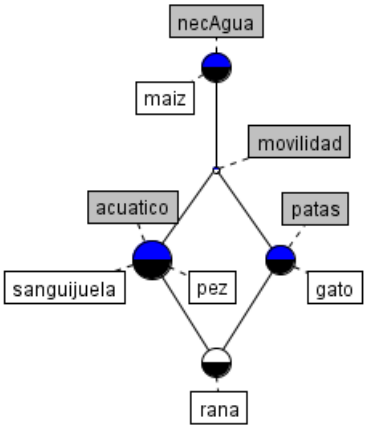
A	B	C	D	E
	necAgua	acuatico	movilidad	patas
gato	X		X	X
sanguijuela	X	X	X	
rana	X	X	X	X
maiz	X			
pez	X	X	X	

En nuestro código de Haskell hemos utilizado las letras `g,s,r,m,p` para referirnos a los respectivos objetos *gato, sanguijuela, rana, maíz* y *pez*. En el caso de los atributos hemos optado por utilizar números; es decir, *necesita agua* será el atributo número 1, *acuático* será el 2, *movilidad* será el 3 y *patas* el 4.

Los diagramas de líneas consisten en círculos, líneas y los nombres de todos los objetos y todos los atributos del contexto dado. Los círculos representan los conceptos y la información del contexto puede ser leída desde el diagrama lineal siguiendo una simple "regla de lectura":

"Un objeto o tiene un atributo a si y sólo si hay un camino hacia arriba que va desde el círculo nombrado como o hasta el círculo nombrado como a."

El retículo de conceptos del ejemplo 2 viene dado por:



Del diagrama de líneas, podemos obtener la siguiente información (utilizando la "regla de lectura" del diagrama):

- La extensión de cada atributo (etiquetas grises) estará formada por el conjunto de todos los objetos localizados desde el círculo correspondiente a dicho atributo y todos los demás objetos que pueden ser alcanzados bajando por una línea desde dicho círculo. La extensión del concepto de la cima es siempre el conjunto de todos los objetos.

- La intensión de cada objeto (etiquetas blancas) será el conjunto de atributos que iremos encontrando al seguir todas las líneas hacia arriba desde el círculo donde se encuentre dicho objeto.

Con todo esto podemos observar lo siguiente:

- El objeto *maíz* sólo tiene el atributo *necesita agua*.
- El objeto *rana* tiene todos los atributos; *necesita agua, movilidad, acuático* y *patas*.
- *Necesita agua* y *acuático* son atributos de los objetos *maíz, sanguijuela, pez* y *rana*.
- etc

Así obtenemos los conceptos
([rana], [necesita agua, movilidad, acuático, patas]),
([rana, sanguijuela, pez], [necesita agua, movilidad, acuático]),
([sanguijuela, pez, gato, rana], [movilidad, necesita agua]),
([rana, gato], [patas, movilidad, necesita agua]) y
([maíz, sanguijuela, pez, gato, rana], [necesita agua])
como bien anticipábamos antes.

Algunas propiedades de la relación de subconcepto son las siguientes:

Proposición 7
La relación $\leq$ (ser subconcepto) cumple las siguientes propiedades:

Reflexiva: *para cualquier concepto (G, M), (G, M) es subconcepto de él mismo.*

Comprobamos en Haskell esta propiedad utilizando una lista por comprensión; es decir, comprobamos que todos los conceptos que provienen del conjunto obtenido a partir de la definición 9 verifican que son subconceptos de ellos mismos.

```
subc_ref :: Contexto Int Int -> Bool
subc_ref c =
     and [esSubconcepto c (gs1,ms1) (gs1,ms1)|
            (gs1,ms1) <- conceptos c]

-- quickCheckWith (stdArgs {maxSize= 30})  subc_ref
-- +++ OK, passed 100 tests.
```

Transitiva: *para cualesquiera conceptos $(G_1, M_1), (G_2, M_2)$ y (G_3, M_3) cumpliéndose que $(G_1, M_1) \leq (G_2, M_2)$ y $(G_2, M_2) \leq (G_3, M_3)$, se tiene que $(G_1, M_1) \leq (G_3, M_3)$.*

De manera análoga, comprobamos la propiedad en Haskell utilizando una lista por comprensión.

```
subc_tran :: Contexto Int Int -> Bool
subc_tran c =
     and [esSubconcepto c (gs1,ms1) (gs3,ms3)|
          (gs1,ms1)<- conceptos c,
          (gs2,ms2)<- conceptos c,
          (gs3,ms3)<- conceptos c,
           esSubconcepto c (gs1,ms1) (gs2,ms2),
           esSubconcepto c (gs2,ms2) (gs3,ms3)]

-- quickCheckWith (stdArgs {maxSize= 20})  subc_tran
-- +++ OK, passed 100 tests.
```

Antisimétrica: *para cualesquiera conceptos* (G_1, M_1) *y* (G_2, M_2) *tales que* $(G_1, M_1) \leq (G_2, M_2)$ *y* $(G_2, M_2) \leq (G_1, M_1)$*, se cumple que* $(G_1, M_1) = (G_2, M_2)$*.*

Lo comprobamos en Haskell de la misma manera que las propiedades anteriores.

```
subc_antisim :: Contexto Int Int -> Bool
subc_antisim c =
     and [(gs1,ms1) == (gs2,ms2)|
          (gs1,ms1)<- conceptos c,
          (gs2,ms2)<- conceptos c,
           esSubconcepto c (gs1,ms1) (gs2,ms2),
           esSubconcepto c (gs2,ms2) (gs1,ms1)]

-- quickCheckWith (stdArgs {maxSize= 20})  subc_antisim
-- +++ OK, passed 100 tests.
```

DEMOSTRACIÓN:

Reflexiva: ¿$(G, M) \leq (G, M)$?

$G \subseteq G$ (ó $M \subseteq M$) $\Rightarrow (G, M) \leq (G, M)$.

Transitiva: $(G_1, M_1) \leq (G_2, M_2)$ y $(G_2, M_2) \leq (G_3, M_3)$,

¿$(G_1, M_1) \leq (G_3, M_3)$?

Por ser $(G_1, M_1) \leq (G_2, M_2)$ se tiene que $G_1 \subseteq G_2$ (ó $M_1 \subseteq M_2$).

Por ser $(G_2, M_2) \leq (G_3, M_3)$ se tiene que $G_2 \subseteq G_3$ (ó $M_2 \subseteq M_3$).

Como $G_1 \subseteq G_2 \subseteq G_3 \Rightarrow G_1 \subseteq G_3$ (ó $M_1 \subseteq M_3$) $\Rightarrow$

$(G_1, M_1) \leq (G_3, M_3)$.

Antisimétrica: $(G_1, M_1) \leq (G_2, M_2)$ y $(G_2, M_2) \leq (G_1, M_1)$,

¿$(G_1, M_1) = (G_2, M_2)$?

Por ser $(G_1, M_1) \leq (G_2, M_2)$ se tiene que $G_1 \subseteq G_2$ (ó $M_1 \subseteq M_2$).

Por ser $(G_2, M_2) \leq (G_1, M_1)$ se tiene que $G_2 \subseteq G_1$ (ó $M_2 \subseteq M_1$).

Pero si $G_1 \subseteq G_2$ y $G_2 \subseteq G_1$ se tiene que $G_1 = G_2$ (ó $M_1 = M_2$) por lo que $(G_1, M_1) = (G_2, M_2)$.

□

Proposición 8
Sea $\mathcal{C} = (O, A, R)$ un contexto. Dados $C_1 = (G_1, M_1)$ y $C_2 = (G_2, M_2)$ dos conceptos de $\mathcal{C}$, se verifica lo siguiente:

▷ *El mayor subconcepto común es*

$$C_1 \wedge C_2 = (G_1 \cap G_2, (M_1 \cup M_2)'')$$

En Haskell quedaría definida de la siguiente forma:

```
mayorSubconcepto :: (Ord a, Ord b) =>
    Contexto a b -> ([a],[b]) -> ([a],[b]) -> ([a],[b])
mayorSubconcepto c (gs1,ms1) (gs2,ms2) =
    (gs1 `intersect` gs2, clausuraA c (ms1 `union` ms2))
```

▷ *El menor superconcepto común es*

$$C_1 \vee C_2 = ((G_1 \cup G_2)'', M_1 \cap M_2)$$

Su definición en Haskell viene dada como sigue:

```
menorSuperconcepto :: (Ord a, Ord b) =>
    Contexto a b -> ([a],[b]) -> ([a],[b]) -> ([a],[b])
menorSuperconcepto c (gs1,ms1) (gs2,ms2) =
    (clausura0 c (gs1 `union` gs2), ms1 `intersect` ms2)
```

Nota 2 Para cualesquiera dos conceptos, existe el menor superconcepto común y el mayor subconcepto común y éstos son únicos.

3.2.3. Generación de todos los conceptos de un contexto formal

Es el momento de presentar un algoritmo para la generación de todos los conceptos de un contexto, como bien anticipábamos antes.

Algoritmo 1

Este algoritmo se basa en dos propiedades principales:

1. Todo concepto de un contexto $\mathcal{C} = (O, A, R)$ es de la forma (Y', Y'') para algún conjunto de atributos Y. Y, recíprocamente, todo par de la forma (Y', Y'') donde Y es un conjunto de atributos de $\mathcal{C}$, es un concepto de $\mathcal{C}$.

2. Para todo conjunto de atributos Y, $Y' = \bigcap_{a \in Y} \{a\}'$.

Pasemos a describir el algoritmo:

Paso 1) Generar el conjunto de todas las posibles extensiones de los conceptos, usando la propiedad 2.

```
candidatosExtensiones1:: (Ord a, Ord b) =>
                         Contexto a b -> [[a]]
candidatosExtensiones1 c = nub (aux ats [obs])
     where ats = atributos c
           obs = objetos c
           aux [] ac = ac
           aux (y:ys) ac =
                aux ys (ac ++ map (intersect y1) ac)
              where y1 = extension c [y]
```

```
-- candidatosExtensiones1_b ejd
-- ["gsrmp","gsrp","gr","srp","r"]
-- (0.02 secs, 0 bytes)
```

Veamos cómo funciona el primer paso con nuestro ejemplo 2, para lo que mostramos la evolución del acumulador `ac`:

Paso 1. $\{O\} = \{\{g, s, r, m, p\}\}$.

Paso 2. $\{\{g, s, r, m, p\}\}$
(pues $\{N\}' = \{g, s, r, m, p\} \bigcap \{g, s, r, m, p\} = \{g, s, r, m, p\}$).

Paso 3. $\{\{g, s, r, m, p\}, \{s, r, p\}\}$
(pues $\{A\}' = \{s, r, p\} \bigcap \{g, s, r, m, p\} = \{s, r, p\}$).

Paso 4. $\{\{g, s, r, m, p\}, \{s, r, p\}, \{g, s, r, p\}\}$
(pues $\{M\}' = \{g, s, r, p\} \bigcap \{g, s, r, m, p\} = \{g, s, r, p\}$,
$\{M\}' \bigcap \{s, r, p\} = \{s, r, p\}$ y $\{M\}' \bigcap \{g, s, r, p\} = \{g, s, r, p\}$).

Paso 5. Por último (ya que en el siguiente paso la lista `ats` será vacía)
$\{\{g, s, r, m, p\}, \{s, r, p\}, \{g, s, r, p\}, \{g, r\}, \{r\}\}$
(pues $\{P\}' = \{g, r\} \bigcap \{g, s, r, m, p\} = \{g, r\}$,
$\{P\}' \bigcap \{s, r, p\} = \{r\}$ y $\{P\}' \bigcap \{g, s, r, p\} = \{g, r\}$).

Podemos mejorar, computacionalmente hablando, la definición anterior en Haskell. Para ello utilizamos un `foldl`.

```
candidatosExtensiones1_b:: (Ord a, Ord b) =>
                           Contexto a b -> [[a]]
candidatosExtensiones1_b c =
            nub(foldl f [objetos c] (atributos c))
  where f ac y = ac ++ map (intersect (extension c [y])) ac

-- candidatosExtensiones1_b ejd
-- ["gsrmp","gsrp","gr","srp","r"]
-- (0.00 secs, 0 bytes)
```

Paso 2) Para cada elemento del conjunto generado en el paso anterior, obtener su intensión.

Haskell posee la función predefinida `map f xs` que nos devuelve la lista obtenida aplicando `f` a cada elemento de `xs`. Por este motivo, podemos hacer uso de ella para conseguir este paso.

```
candidatosIntensiones1:: (Ord a, Ord b) =>
                         Contexto a b -> [[b]]
candidatosIntensiones1 c =
     map (intension c) (candidatosExtensiones1 c)

-- candidatosIntensiones1 ejd
-- [[1],[1,3],[1,3,4],[1,3,2],[1,3,4,2]]
```

Paso 3) Obtenemos el conjunto de todos los conceptos del contexto, formando cada uno de ellos de la siguiente forma: (Y', Y'').

Cada Y' será cada uno de los elementos del conjunto obtenido en el primer paso del algoritmo, y cada Y'' será cada uno de los elementos del conjunto del paso dos. Por este motivo, sólo nos basta con usar la función predefinida `zip xs ys` que devuelve la lista de pares formada por los correspondientes elementos de `xs` e `ys`.

```
generaConceptos1::(Ord a, Ord b) =>
                  Contexto a b -> [([a], [b])]
generaConceptos1 c =
  zip (candidatosExtensiones1 c) (candidatosIntensiones1 c)

-- generaConceptos1 ejd
-- [("gsrmp",[1]),("gsrp",[1,3]),("gr",[1,3,4]),
-- ("srp",[1,3,2]),("r",[1,3,4,2])]
```

Podríamos interpretar este resultado como sigue:

- ({g,s,r,m,p},{N}), es el concepto de los seres vivos.
- ({g,s,r,p},{N,M}), es el concepto de los animales.

- ({g,r},{N,M,P}), es el concepto de los animales con patas.
- ({s,r,p},{N,A,M}), es el concepto de los animales acuáticos.
- ({r}{N,A,M,P}), es el concepto de los anfibios.

Con este algoritmo obtenemos el conjunto de todos los conceptos de un contexto de una manera mucho más rápida que con la obtención de dicho conjunto utilizando la definición 9. Comprobemos que ambos generan los mismos conceptos.

Proposición 9
El conjunto de los conceptos generados en el algoritmo 1 es el mismo conjunto que el obtenido en la generación de conceptos de la definición 9.

Para comprobar esta proposición definamos en Haskell la igualdad de conjuntos.

```
igualConjunto xs ys = subconjunto xs ys &&
                      subconjunto ys xs &&
                      length xs == length ys
```

Pasemos ahora a comprobar la propiedad.

```
generaConceptos_prop1:: Contexto Int Int -> Bool
generaConceptos_prop1 c =
     igualConjunto (generaConceptos1 c) (conceptos c)

-- quickCheckWith (stdArgs {maxSize= 20})
                    generaConceptos_prop1
-- +++ OK, passed 100 tests.
```

De forma análoga al algoritmo anterior, podemos generar otro algoritmo de obtención de conceptos pero comenzando a buscar las intensiones del contexto en el paso 1.

En general, la generación de conceptos del algoritmo 1 será mejor que la generación de conceptos mediante el algoritmo 2 pues lo normal es que el número de atributos sea menor que el número de objetos y por este motivo será más costoso el primer paso en el segundo algoritmo.

Algoritmo 2

Este segundo algoritmo se basa en los siguientes puntos:

1. Todo concepto del contexto $\mathcal{C} = (O, A, R)$ es de la forma (X'', X') para algún conjunto de objetos X. Y, recíprocamente, todo par de la forma (X'', X') donde X es un conjunto de objetos de $\mathcal{C}$, es un concepto de $\mathcal{C}$.

2. Para todo conjunto de objetos X, $X' = \bigcap_{b \in X} \{b\}'$.

Pasemos a enumerar los pasos del algoritmo. Este segundo algoritmo tiene el mismo procedimiento que el primero.

Paso 1) Generar el conjunto de todas las posibles intensiones de los conceptos, usando el paso 2.

De la misma manera que en el paso 1 del algoritmo primero, definimos en Haskell el paso 1 del segundo algoritmo. La única diferencia es que en el lugar de la lista de los objetos ahora estará la lista de los atributos y viceversa.

```
candidatosIntensiones2:: (Ord a, Ord b) =>
                         Contexto a b -> [[b]]
candidatosIntensiones2 c = nub (aux obs [ats])
     where ats = atributos c
           obs = objetos c
           aux [] ac = ac
           aux (x:xs) ac =
                aux xs (ac ++ map (intersect x1) ac)
              where x1 = intension c [x]

-- candidatosIntensiones2 ejd
-- [[1,3,4,2],[1,3,4],[1,3,2],[1,3],[1]]
-- (0.02 secs, 0 bytes)
```

Paso 2) Para cada elemento del conjunto generado en el paso anterior, obtener su extensión.

```
candidatosExtensiones2:: (Ord a, Ord b) =>
                         Contexto a b -> [[a]]
candidatosExtensiones2 c =
     map (extension c) (candidatosIntensiones2 c)

-- candidatosExtensiones2 ejd
-- ["r","gr","srp","gsrp","gsrmp"]
-- (0.00 secs, 0 bytes)
```

Paso 3) Obtenemos el conjunto de todos los conceptos del contexto, formando el concepto correspondiente; (X'', X').

En este paso, cada X'' será cada uno de los elementos del conjunto generado en el paso 2 y cada X' será cada elemento del conjunto del paso 1.

```
generaConceptos2::(Ord a, Ord b) =>
                  Contexto a b -> [([a], [b])]
generaConceptos2 c =
  zip (candidatosExtensiones2 c) (candidatosIntensiones2 c)

-- generaConceptos2 ejd
-- [("r",[1,3,4,2]),("gr",[1,3,4]),("srp",[1,3,2]),
-- ("gsrp",[1,3]),("gsrmp",[1])]
-- (0.02 secs, 0 bytes)
```

Por similitud, tenemos la misma proposición que con el algoritmo 1 pero ahora, con el 2.

Proposición 10
El conjunto de los conceptos generados en el algoritmo 2 es el mismo conjunto que el obtenido en la generación de conceptos de la definición 9.

```
generaConceptos_prop2:: Contexto Int Int -> Bool
generaConceptos_prop2 c =
     igualConjunto (generaConceptos2 c) (conceptos c)
```

```
-- quickCheckWith (stdArgs {maxSize= 20})
                    generaConceptos_prop2
-- +++ OK, passed 100 tests.
```

Capítulo 4

Razonamiento en contextos formales

4.1. Implicaciones entre atributos

Una vez que tenemos claro las definiciones de contexto y concepto, sería interesante indagar sobre las interacciones entre los atributos. Esto se debe a que normalmente se estudia la clasificación de un gran número de objetos con respecto a pocos atributos lo que imposibilita "dibujar" el retículo de los conceptos. Por ello, nos interesa disponer de reglas que expresen relaciones entre los atributos del contexto; por ejemplo, "si un objeto o posee los atributos $a_1, \ldots, a_n$, también posee los atributos $b_1, \ldots, b_m$". Para ello utilizaremos las relaciones lógicas que encontraremos entre los atributos aunque hay que tener en cuenta que las relaciones lógicas que se obtengan son aquellas que vengan confirmadas por nuestro contexto, esto es, que podría representar un conocimiento concreto en un instante de tiempo determinado pero que podría variar al añadir un mayor número de objetos o atributos.

Definición 11 Sea un contexto (O, A, R). Una **implicación** entre atributos es un par de subconjuntos, M y N, del conjunto de atributos A, denotado por $M \to N$.

En nuestro código de Haskell, representaremos las implicaciones $M \to N$ como pares `(m,n)`.

```
esImplicacion :: (Ord a, Ord b) =>
                 Contexto a b -> ([b],[b]) -> Bool
esImplicacion c (m,n) = subconjunto m (atributos c) &&
                        subconjunto n (atributos c)
```

Por lo tanto, el conjunto de todas las implicaciones de un contexto $\mathcal{C}$ vendrá dado por el conjunto de todas las implicaciones de la forma $M \to N$ en las que M y N serán subconjuntos del conjunto A de atributos.

```
imps :: (Ord a, Ord b) => Contexto a b -> [([b],[b])]
imps c = [(m,n) | m <- atbs, n <- atbs]
         where atbs = subsequences (atributos c)
```

Dejemos claro que, esta definición de implicación es sólo una definición sintáctica; es decir, no se le está dando ningún valor o significado a una implicación. La semántica se le dará en las definiciones posteriores.

Definición 12 Sea (O, A, R) un contexto. Un subconjunto T de A **respeta** o **es modelo de** una implicación $M \to N$ si $M \not\subseteq T$ ó $N \subseteq T$; es decir, si $M \subseteq T$, **entonces también** $N \subseteq T$.

```
esModelo :: (Ord a, Ord b) =>
            Contexto a b -> ([b],[b]) -> [b] -> Bool
esModelo c (m,n) ts = subconjunto ts (atributos c) &&
                (not (subconjunto m ts) || subconjunto n ts)
```

A partir de esta definición podemos obtener el conjunto de modelos de una implicación para un contexto dado $\mathcal{C}$.

```
modelosImp :: (Ord a, Ord b) =>
              Contexto a b -> ([b],[b]) -> [[b]]
modelosImp c (m,n) =  filter (esModelo c (m,n)) (atbs c)
```

Observemos que para esta definición hemos usado la función predefinida de Haskell `filter p f`. Esta función comprueba para cada elemento del segundo dato de entrada (en este caso `atbs c`) si se cumple la propiedad `p`. En caso afirmativo este elemento será incluido en la lista y en caso negativo no.

Definición 13 Dado un contexto (O, A, R). Un subconjunto T de A **respeta** o **es modelo para un conjunto** $\mathcal{L}$ de implicaciones si T es modelo de cada implicación de $\mathcal{L}$.

```
esModeloConj :: (Ord a, Ord b) =>
                Contexto a b -> [([b],[b])] -> [b] -> Bool
esModeloConj c ls ts =
    and [esModelo c (m,n) ts | (m,n) <- ls]
```

Dados un contexto $\mathcal{C}$ y un conjunto de implicaciones $\mathcal{L}$ del contexto podemos obtener, a partir de la definición anterior, el conjunto de modelos de dicho conjunto de implicaciones.

```
modelosConjImp :: (Ord a, Ord b) =>
                  Contexto a b -> [([b],[b])] -> [[b]]
modelosConjImp c ls = filter (esModeloConj c ls) (atbs c)
```

Definición 14 Si (O, A, R) es un contexto, una implicación $M \to N$ es **válida** en un conjunto $\{T_1, T_2, \ldots\}$ de subconjuntos de A si cada uno de los subconjuntos T_i es modelo de la implicación $M \to N$.

```
implValidaConj :: (Ord a, Ord b) =>
                  Contexto a b -> ([b],[b]) -> [[b]] -> Bool
implValidaConj c (m,n) tss = all (esModelo c (m,n)) tss
```

La siguiente definición es aquella que le dará el significado o valor semántico a una implicación en un contexto dado.

Definición 15 Una implicación $M \to N$ es **válida en un contexto** (O, A, R) si ésta es válida en el conjunto de intensiones del conjunto de los objetos.

```
implValida :: (Ord a, Ord b) =>
              Contexto a b -> ([b],[b]) -> Bool
implValida c (m,n) =
    implValidaConj c (m,n) [intension c [o] | o <- objetos c]
```

Luego, el conjunto de todas las implicaciones válidas del contexto $\mathcal{C}$ vendrá dado por:

```
impsValidas :: (Ord a, Ord b) => Contexto a b -> [([b],[b])]
impsValidas c = filter (implValida c) (imps c)
```

Para comprobar si una implicación es válida en un contexto, tenemos que realizar muchos cálculos si seguimos las pautas de la definición anterior. Pero hay otra manera mucho más sencilla de comprobarlo.

Proposición 11
Una implicación $M \to N$ es válida en (O, A, R) si y sólo si $N \subseteq M''$; es decir, todos los atributos de N también son atributos de la clausura de M.

```
prop_10 :: Contexto Int Int -> Bool
prop_10 c = and [subconjunto n (clausuraA c m) |
                    (m,n) <- impsValidas c]

-- quickCheckWith (stdArgs {maxSize= 10}) prop_10
-- +++ OK, passed 100 tests.
```

Introduzcamos la definición de familia cerrada de conjuntos que nos hará falta para la próxima proposición.

Definición 16 Se dice que una familia de conjuntos U es **cerrado** respecto de G si $G \in U$ y U es cerrado bajo intersección; es decir, si
$X \subseteq U \Rightarrow (\cap X) \in U$.

```
cerrado :: Eq a => [a] -> [[a]]  -> Bool
cerrado gs uss = gs `elem` uss &&
                    and [(interseccionG xs) `elem` uss |
                            xs <- tail (subsequences uss)]
```

Proposición 12
Dado un contexto (O, A, R)

a) Si $\mathcal{L}$ es un conjunto de implicaciones en A,

$$\mathfrak{h}(\mathcal{L}) := \{X \subseteq M \mid X \text{ es modelo para un conjunto } \mathcal{L}\}$$

es cerrado respecto del conjunto de atributos A.

b) Si $\mathcal{L}$ es el conjunto de todas las implicaciones válidas de un contexto, $\mathfrak{h}(\mathcal{L})$ es el sistema de todas las intensiones.

Comprobemos con QuickCheck que se cumplen ambos apartados.

Apartado primero:

```
prop_11_a :: Contexto Int Int -> [([Int],[Int])] -> Bool
prop_11_a c ls =
           and [cerrado (atributos c) (modelosConjImp c ls) |
                                ls <- subsequences (imps c)]

-- quickCheckWith (stdArgs {maxSize=2}) prop_11_a
-- +++ OK, passed 100 tests.
-- (0.06 secs, 0 bytes)
-- quickCheckWith (stdArgs {maxSize=3}) prop_11_a
-- +++ OK, passed 100 tests.
-- (0.08 secs, 28177144 bytes)
```

Observemos que para la comprobación hemos exigido que quickCheck tome ejemplos con conjuntos de objetos y atributos con pocos elementos. Esto es debido a que esta propiedad, así enunciada, tiene un gran coste computacional porque tiene que hacer `subsequences (imps c)`.

Veamos el segundo apartado:

```
prop_11_b :: Contexto Int Int -> Bool
prop_11_b c = igualConjunto (modelosConjImp c ls)
                            (candidatosIntensiones2 c)
              where ls = impsValidas c

-- quickCheckWith (stdArgs {maxSize=10}) prop_11_b
-- +++ OK, passed 100 tests.
-- (6.15 secs, 1683378216 bytes)
```

Para este apartado, volvemos a exigirle que tome ejemplos de pequeño tamaño pues los cálculos son pesados y el tiempo se incrementa mucho con ejemplos aleatorios de grandes cantidades de objetos y atributos.

4.2. Base de implicaciones

Llegados a este punto, se nos plantea un nuevo problema pues puede ocurrir que el conjunto de todas las implicaciones válidas en un contexto sea excesivamente grande y/o puede contener muchas implicaciones triviales o redundantes.

Para ello, interesa disponer de un conjunto reducido de implicaciones, a partir de las cuales se **deducirán** todas las demás.

Definamos la noción de consencuencia.

Definición 17 Dado un contexto (O, A, R) se dice que una implicación $M \to N$ **es consecuencia** de un conjunto $\mathcal{L}$ de implicaciones entre atributos si cada subconjunto de A que es modelo del conjunto $\mathcal{L}$ también es modelo de $M \to N$.

```
esConsecuencia :: (Ord a, Ord b) =>
        Contexto a b -> ([b],[b]) -> [([b],[b])] -> Bool
esConsecuencia c (m,n) ls =
    and [esModelo c (m,n) ts | ts <- atbs c,
                               esModeloConj c ls ts]
```

A partir de esta definición podemos obtener el conjunto de todas las implicaciones del contexto $\mathcal{C}$ que son consecuencia del conjunto de implicaciones $\mathcal{L}$.

```
sonConsecuencia :: (Ord a, Ord b) =>
                   Contexto a b -> [([b],[b])] -> [([b],[b])]
sonConsecuencia c ls = [l | l <- imps c,
                            esConsecuencia c l ls]
```

Definición 18 Para un contexto $\mathcal{C} = (O, A, R)$, un conjunto de implicaciones $\mathcal{L}$ se llama **adecuado** en el contexto $\mathcal{C}$, si todas las implicaciones de $\mathcal{L}$ son válidas en $\mathcal{C}$.

```
adecuadoImpl :: (Ord a, Ord b) =>
                Contexto a b -> [([b],[b])] -> Bool
adecuadoImpl c ls = and [implValida c l | l <- ls]
```

Definición 19 Un conjunto $\mathcal{L}$ de implicaciones de un contexto (O, A, R) se llama **completo** si cada implicación válida de (O, A, R) es consecuencia de $\mathcal{L}$.

```
completo :: (Ord a, Ord b) =>
            Contexto a b -> [([b],[b])] -> Bool
completo c ls = and [esConsecuencia c l ls |
                         l <- impsValidas c]
```

Definición 20 Un conjunto $\mathcal{L}$ de implicaciones de un contexto (O, A, R) se llama **no redundante** si ninguna de las implicaciones de $\mathcal{L}$ es consecuencia de las otras.

```
noRedundante :: (Ord a, Ord b) =>
                Contexto a b -> [([b],[b])] -> Bool
noRedundante c ls = [l | l <- ls,
                     esConsecuencia c l (delete l ls)] == []
```

Como anticipábamos más arriba, sería conveniente obtener un conjunto de implicaciones válidas que caracterice todas las implicaciones válidas en un contexto. Por este motivo, buscaremos un conjunto de implicaciones con las propiedades:

Adecuación: todas las implicaciones del conjunto son válidas.

Completitud: toda implicación válida es consecuencia del conjunto.

Irredundancia: no se puede eliminar ninguna de las implicaciones del conjunto.

4.3. Cálculo de la base Stem

El problema ahora es encontrar un conjunto adecuado, completo y no redundante de implicaciones, cuando el conjunto de los atributos es finito. Es decir, queremos encontrar una **base** de implicaciones.

Comencemos definiendo la noción de pseudo-intensión.

Definición 21 Dado un contexto $\mathcal{C} = (O, A, R)$, se dirá que un subconjunto $P \subseteq A$ es una **pseudo-intensión** de $\mathcal{C}$ si se verifica lo siguiente:

- $P \neq P''$.
- Para toda pseudo-intension Q tal que $Q \subset P$, se verifica que $Q'' \subseteq P$.

```
pseudoInt :: (Ord a, Ord b) => Contexto a b -> [b] -> Bool
pseudoInt c [] = [] /= (clausuraA c [])
pseudoInt c p = not (igualConjunto p (clausuraA c p)) &&
                      and [ subconjunto (clausuraA c q) p
                                | q <- init (subsequences p),
                                        pseudoInt c q ]

-- pseudoInt ejd []
-- True
-- pseudoInt ejd [3]
-- False
```

En el contexto del ejemplo 2, $\emptyset$ es una pseudo-intensión, pues
$\emptyset'' = \{g, s, r, m, p\}' = \{N\} \neq \emptyset$
Pero $\{M\}$ no es una pseudo-intensión, pues

- $\{M\}'' \neq \{M\}$ pero
- el único subconjunto propio de M es $\emptyset$, que es una pseudo-intensión y $\emptyset'' \not\subseteq \{M\}$.

A continuación se describe un algoritmo recursivo para la generación de todas las pseudo-intensiones de un contexto, de forma escalonada según el cardinal:

Paso 0: Obtener las pseudo-intensiones del contexto $\mathcal{C}$ de cardinal 0:

$S_0 = \{\emptyset\}$ si $\emptyset$ es pseudo-intensión ó $S_0 = \{\}$ en otro caso.

Paso $k+1$: Obtener S_{k+1} (pseudo-intensiones de cardinal $\leq k+1$) a partir de S_k (pseudo-intensiones de cardinal $\leq k$).

Para este propósito vamos a definir la noción de "pseudo-intensión restringida" que nos servirá para calcular todas las pseudo-intensiones de cardinal $\leq k+1$ a partir de todas las pseudo-intensiones de cardinal $\leq k$ que en el paso $k+1$ ya estarán calculadas.

Entonces, "P es una **pseudo-intensión restringida a** S" si se verifica lo siguiente:

- $P \neq P''$
- $\forall X \in S$, si $X \subset P \Rightarrow X'' \subseteq P$

$$S_{k+1} = S_k \bigcup \{P \subseteq A \text{ tal que } |P| = k+1, \\ P \text{ es una pseudo-intensión restringida a } S_k\}$$

Finalmente, obtendremos las pseudo-intensiones de un contexto $\mathcal{C} = (O, A, R)$ como $S_{|A|}$.

```
subconjuntosCard:: [a] -> Int -> [[a]]
subconjuntosCard _ 0      = [[]]
subconjuntosCard [] _     = []
subconjuntosCard (x:xs) k =
    map (x:) (subconjuntosCard xs (k-1)) ++
                            subconjuntosCard xs k

pseudoIntRes :: (Ord a, Ord b) =>
                Contexto a b -> [b] -> [[b]] -> Bool
pseudoIntRes c [] ss = [] /= (clausuraA c [])
pseudoIntRes c p ss =
    not (igualConjunto p (clausuraA c p)) &&
        and [subconjunto (clausuraA c q) p | q <- cs]
            where cs = intersect ss (init (subsequences p))

pseudoIntensionesCard :: (Ord a, Ord b) =>
                         Contexto a b -> Int -> [[b]]
pseudoIntensionesCard c 0 | [] == (clausuraA c []) = []
                          | otherwise              = [[]]
pseudoIntensionesCard c k =
    union ss [p | p <- cs, pseudoIntRes c p ss]
     where ss = pseudoIntensionesCard c (k-1)
           cs = subconjuntosCard (atributos c) k
```

```
pseudoIntensiones :: (Ord a, Ord b) => Contexto a b -> [[b]]
pseudoIntensiones c = pseudoIntensionesCard c n
    where n = length (atributos c)

-- pseudoIntensiones ejd
-- [[],[1,4],[1,2]]
```

Una vez obtenidas todas las pseudo-intensiones del contexto podemos formar el conjunto base que buscábamos.

Definición 22 Sea $\mathcal{C} = (O, A, R)$ un contexto. El conjunto

$$\{P \longrightarrow P'' : P \text{ es una pseudo-intensión del contexto}\}$$

es una base, denominada **base Stem** o **base de Duquenne-Guigues**.

Nota 3 En la práctica, se toman las implicaciones $P \longrightarrow (P'' \setminus P)$.

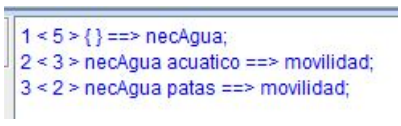
Pasemos a dar la definición en Haskell de la base Stem. Lo haremos fácilmente mediante una lista por comprensión, recorriendo todas las pseudo-intensiones del contexto.

```
diferenciaL :: Eq a => [a] -> [a] -> [a]
diferenciaL ys xs = [y | y <- ys, y `notElem` xs]

baseStem :: (Ord a, Ord b) => Contexto a b -> [([b],[b])]
baseStem c = [(p, diferenciaL (clausuraA c p) p) |
                p <- pseudoIntensiones c]

-- baseStem ejd
-- [([],[1]),([1,4],[3]),([1,2],[3])]
```

También podemos obtener la base Stem del ejemplo 2 mediante el programa conexp. Este programa nos muestra la base de la siguiente forma:



Por último, comprobemos que la base Stem cumple las propiedades que nombrábamos anteriormente.

Teorema 1
Sea $\mathcal{C} = (O, A, R)$ un contexto. El conjunto de implicaciones

$$\mathcal{L} = \{P \longrightarrow P''\backslash\{P\} : P \text{ es una pseudo-intensión}\}$$

es adecuado, no redundante y completo.

```
teorema_1 :: Contexto Int Int -> Bool
teorema_1 c = adecuadoImpl c ls &&
              noRedundante c ls &&
              completo c ls
             where ls = baseStem c

-- quickCheckWith (stdArgs {maxSize=30}) teorema_1
-- +++ OK, passed 100 tests.
```

DEMOSTRACIÓN:

• Probemos que $\mathcal{L}$ es adecuado.

Para ver que $\mathcal{L}$ es adecuado tenemos que ver que se cumple que todas las implicaciones de $\mathcal{L}$ son válidas en el contexto. Ahora bien, una implicación $M \longrightarrow N$ es válida en un contexto $\mathcal{C}$ si y sólo si $N \subseteq M''$.

Todas las implicaciones de $\mathcal{L}$ son de la forma $P \longrightarrow P''\backslash\{P\}$ y por tanto, se cumple trivialmente que $P''\backslash\{P\} \subseteq (P)'' = P''$.

$\Longrightarrow \mathcal{L}$ es adecuado.

• Veamos que $\mathcal{L}$ es no redundante.

Para ver que $\mathcal{L}$ es no redundante tenemos que probar que para toda $I = P \longrightarrow P''\backslash\{P\}$, implicación de $\mathcal{L}$, se verifica que I no es consecuencia de $\mathcal{L}\backslash\{I\}$; es decir, existe una implicación de $\mathcal{L}\backslash\{I\}$ que no es modelo de I.

Tomemos P y probemos lo siguiente:

1. P es modelo de $\mathcal{L}\backslash\{P \longrightarrow P''\backslash\{P\}\}$.
2. P no es modelo de $P \longrightarrow P''\backslash\{P\}$.

1) Para probar que P es modelo de $\mathcal{L}\backslash\{P \longrightarrow P''\backslash\{P\}\}$, tengo que probar que $\forall Q \longrightarrow Q''\backslash\{Q\}$ implicación de $\mathcal{L}\backslash\{P \longrightarrow P''\backslash\{P\}\}$, se cumple que P es modelo de $Q \longrightarrow Q''\backslash\{Q\}$.

   - Si $Q \not\subseteq P$ ya lo tenemos.
   - Si $Q \subseteq P \Rightarrow Q'' \subseteq P$ por ser P pseudo-intensión. Por tanto, ya tenemos que $Q''\backslash\{Q\} \subseteq P$. Luego, P es modelo de $Q \longrightarrow Q''\backslash\{Q\}$.

   Por tanto ya tenemos probado el primer punto.

2) Probemos que P no es modelo de $P \longrightarrow P''\backslash\{P\}$.

   La condición $P \not\subseteq P$ no se cumple y la condición $P''\backslash\{P\} \subseteq P$ tampoco se cumple pues sería, $P \subseteq P''$ por la definición de clausura por tanto nunca se puede dar que $P'' \subseteq P$ pues $P \neq P''$ por ser pseudo-intensión.

   Por tanto, ya tenemos probada el segundo punto y con éste, hemos probado que $\mathcal{L}$ es no redundante.

- Por último demostremos que $\mathcal{L}$ es completo.

Para probar que $\mathcal{L}$ es completo tenemos que probar que toda implicación $M \longrightarrow N$ válida en el contexto $\mathcal{C}$ es consecuencia de $\mathcal{L}$. Es decir, que si $T \subseteq A$ y T es modelo de $\mathcal{L}$, entonces T es modelo de $M \longrightarrow N$. Lo probamos distinguiendo dos casos:

Caso 1) $T = T''$.

Supongamos que $M \subseteq T$, entonces aplicando dos veces el apartado 2 de la proposición 3 se tiene que $M'' \subseteq T''$. Por otra parte, por ser $M \longrightarrow N$ implicación válida del contexto se cumple que $N \subseteq M''$. Luego $N \subseteq M'' \subseteq T'' = T$; $N \subseteq T$.

Por tanto, si $T = T''$, T es modelo de $M \longrightarrow N$.

Caso 2) $T \neq T''$.

En primer lugar, comprobamos que si T es modelo de $\mathcal{L}$ y $T \neq T''$, entonces T es una pseudo-intensión. En efecto, es suficiente probar que se verifica la segunda condición: sea Q una pseudo-intensión tal que $Q \subset T$. Por ser pseudo-intensión, $Q \longrightarrow Q''\backslash\{Q\} \in \mathcal{L}$ y, por tanto, T es modelo de $Q \longrightarrow Q''\backslash\{Q\}$. Luego $Q'' \subseteq T$.

Entonces, $T \longrightarrow T''\backslash\{T\} \in \mathcal{L}$. Como T es modelo de $\mathcal{L}$, T es modelo de $T \longrightarrow T''\backslash\{T\}$ y, por tanto, $T'' \subseteq T$. Como siempre

se verifica que $T \subseteq T''$, tendríamos que $T = T''$, en contradicción con el supuesto de este apartado.

Por tanto, $T \neq T''$ que nos lleva al caso 1.

Por tanto, $\mathcal{L}$ es completo.

□

A pesar de que hemos dicho a lo largo de todo el texto que cualquiera de las representaciones usadas en Haskell eran válidas, algunas son más eficientes que otras. Para finalizar este trabajo, comparemos el tiempo que tarda Haskell en calcular la base Stem de nuestro ejemplo siguiente con cada una de ellas:

```
ej5 = creaContexto [('a', [0..10]),
                    ('b', [1..9]),
                    ('c', [1..5]),
                    ('d', [1..10]),
                    ('e', [0..4]),
                    ('f', [1..5])]
```

ContextoConListas: `(1.70 secs, 503522272 bytes)`.

ContextoConFunciones: `(15.01 secs, 2647719728 bytes)`.

ContextoConMatrices: `(38.08 secs, 9637173536 bytes)`.

ContextoConDiccionarios: `(1.78 secs, 532461096 bytes)`.

Podemos observar que las representaciones más eficiente son la de listas y la de diccionarios (prácticamente igual de eficientes); siendo la de matrices la más lenta de todas.

Bibliografía

[1] Ganter, B. y Wille, R. *Formal Concept Analysis*. Springer–Verlag, 1999.

[2] *The Concept Explorer*. Ver `http://conexp.sourceforge.net`.

[3] *The Haskell Programming Language*. Ver `https://wiki.haskell.org/Haskell`.

[4] *¡Aprende Haskell por el bien de todos!* Ver `http://aprendehaskell.es`.

[5] *Análisis Formal de Conceptos de Fernando Sancho*. Ver `http://www.cs.us.es/ fsancho/?e=78`.

[6] *Historia de Haskell*. Ver `https://es.wikipedia.org/wiki/Haskell#Historia`.

[7] José A. Alonso, J. Borrego, M.J. Hidalgo, F.J. Martín y J.L. Ruíz. *Una introducción al Análisis Formal de Conceptos en PVS*. Ver `http://www.cs.us.es/ jalonso/pub/2002-IDEIA-AFC.pdf`.

[8] M.J. Hidalgo y J. Borrego *Curso sobre Análisis Formal de Conceptos*.

[9] José A. Alonso. *Transparencias de la asignatura de Informática*. Ver `https://www.cs.us.es/ jalonso/cursos/i1m/temas.php`.

[10] *Pruebas automáticamente aleatorias con QuickCheck*. Ver `https://www.schoolofhaskell.com/user/XookDo/introduccion-a-la-programacion-funcional/parte-8/tutorial`.

[11] Koen Claessen y John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. Proc. Of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000, Revisado en 2006-01-29.

[12] *QuickCheck: Automatic testing of Haskell programs*. Ver `https://hackage.haskell.org/package/QuickCheck`.

[13] *Formal Concept Analysis Homepage*. Ver `http://www.upriss.org.uk/fca/fca.html`.

[14] Thomas Sutton. *A Complete Idiot's Introduction to Formal Concept Analysis for Dummies to Teach Themselves*. Ver `http://code.ouroborus.net/fp-syd/past/2013/2013-11-Sutton-ConceptAnalysis.pdf`.

[15] *Thomas Sutton's code*. Ver `https://github.com/thsutton/fca/blob/master/src/Main.hs`.

[16] *Formal concept analysis*. Ver `https://en.wikipedia.org/wiki/Formal_concept_analysis`.

Apéndice

Representaciones como TADs

TAD con listas

```
module ContextoConListas
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.List

-- Representación elegida:
data Contexto a b = C [a] [b] [(a,[b])]
                    deriving (Eq, Show)


creaContexto :: (Eq a, Eq b) => [(a,[b])] -> Contexto a b
creaContexto ps = C as bs ps
    where as = map fst ps
          bs = nub $ concatMap snd ps

-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs
```

```
relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C _ _ rs) = concatMap f rs
    where f (x,ys) = [(x,y) | y <-ys]
```

TAD con funciones

```
module ContextoConFunciones
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.List

-- Representación elegida:
data Contexto a b = C [a] [b] (a -> b -> Bool)


creaContexto :: (Eq a, Eq b) => [(a,[b])] -> Contexto a b
creaContexto ps = C as bs f
    where as = map fst ps
          bs = nub $ concatMap snd ps
          f x y = x `elem` as && y `elem` img ps x

img ps x = head [ys | (x',ys) <- ps, x' == x]

-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C as bs f) = [(x,y) | x <- as, y <- bs, f x y]
```

TAD con matrices

```
module ContextoConMatrices
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.Matrix
import Data.List

-- Representación elegida:
data Contexto a b = C [a] [b] (Matrix Int)
                  deriving (Eq, Show)


creaContexto :: (Eq a, Eq b, Ord b) =>
                [(a,[b])] -> Contexto a b
creaContexto ps = C as bs m
     where as = map fst ps
           bs = sort $ nub $ concatMap snd ps
           m = deListaParesAmatriz ps

deListaParesAmatriz :: (Eq a, Eq b, Ord b) =>
                       [(a,[b])] -> Matrix Int
deListaParesAmatriz ps = matrix m n f
    where as = map fst ps
          m = length as
          bs = sort $ nub $ concatMap snd ps
          n = length bs
          relacionado ps x y = x `elem` as &&
                               y `elem` img ps x
          f (i,j)| relacionado ps (as !! (i-1)) (bs !! (j-1))
                                  = 1
                 | otherwise      = 0

img ps x = head [ys | (x',ys) <- ps, x' == x]
```

```
-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C as bs m) = [(x,y) | x <- as, y <- bs,
                          getElem (posicion x as)
                                  (posicion y bs) m == 1]

posicion x xs = head [y | (x',y) <- zip xs [1..], x == x']
```

TAD con diccionarios

```
module ContextoConDiccionarios
    (Contexto,
     creaContexto,
     objetos,
     atributos,
     relaciones
    ) where

import Data.List
import Data.Ix
import qualified Data.Map as M

-- Representación elegida:
data Contexto a b = C [a] [b] (M.Map a [b])
                  deriving (Eq, Show)


creaContexto :: (Ord a, Eq b) => [(a,[b])] -> Contexto a b
creaContexto ps = C as bs (foldr f M.empty ps)
    where as = map fst ps
          bs = nub $ concatMap snd ps
          f (x,ys) d = M.insert x ys d

-- Funciones de acceso:
objetos:: (Eq a, Eq b) => Contexto a b -> [a]
objetos (C as _ _) = as

atributos:: (Eq a, Eq b) => Contexto a b -> [b]
atributos (C _ bs _) = bs

relaciones:: (Eq a, Eq b) => Contexto a b -> [(a,b)]
relaciones (C _ _ ds) = concatMap f (M.toList ds)
    where f (x,ys) = [(x,y) | y <- ys]
```

Análisis Formal de Conceptos

```
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

import Test.QuickCheck
import ContextoConListas
-- import ContextoConFunciones
-- import ContextoConMatrices
-- import ContextoConDiccionarios

import Data.List

ej1, ej2, ej3, ejd,ej5 :: Contexto Char Int
ej1 = creaContexto [('a', [0,2]),
                    ('b', [1,2]),
                    ('c', [1,2]),
                    ('d', [1,2]),
                    ('e', [0]),
                    ('f', [1])]

ej2 = creaContexto [('a', [0..100]),
                    ('b', [1..100]),
                    ('c', [1..20]),
                    ('d', [1..100]),
                    ('e', [0..100]),
                    ('f', [1..50])]

ej3 = creaContexto [('a', [0..20]),
                    ('b', [1..20]),
                    ('c', [1..20]),
                    ('d', [1..20]),
                    ('e', [0..10]),
                    ('f', [1..5])]

ejd = creaContexto [('g', [1,3,4]),
                    ('s', [1,2,3]),
                    ('r', [1,2,3,4]),
                    ('m', [1]),
                    ('p', [1,2,3])]
```

```
ej5 = creaContexto [('a', [0..10]),
                    ('b', [1..9]),
                    ('c', [1..5]),
                    ('d', [1..10]),
                    ('e', [0..4]),
                    ('f', [1..5])]


atributosElto :: (Eq a, Eq b) =>
                 Contexto a b -> a -> [b]
atributosElto c o = [a | a <- atributos c,
                         (o,a) `elem` relaciones c]

-- atributosElto ejd 'g'
-- [1,3,4]
-- atributosElto ejd 's'
-- [1,3,2]


objetosElto :: (Eq a, Eq b) => Contexto a b -> b -> [a]
objetosElto c a = [o | o <- objetos c,
                       (o,a) `elem` relaciones c]

-- objetosElto ejd 1
-- "gsrmp"
-- objetosElto ejd 4
-- "gr"


obss :: (Eq a, Eq b) => Contexto a b -> [[a]]
obss = subsequences . objetos


atbs :: (Eq a, Eq b) => Contexto a b -> [[b]]
atbs = subsequences . atributos
```

```
relacionado:: (Eq a, Eq b) => Contexto a b -> a -> b -> Bool
relacionado c o a = (o,a) `elem` relaciones c

-- relacionado ejd 'g' 4
-- True
-- relacionado ejd 'g' 2
-- False


intension:: (Eq a, Eq b) => Contexto a b -> [a] -> [b]
intension c gs =
    [a | a <- atributos c, all (\g -> relacionado c g a ) gs]

-- atributos ej1       == [0,2,1]
-- intension ej1 "b"   == [2,1]
-- intension ej1 "bcd" == [2,1]
-- intension ej1 "aec" == []

-- intension ejd "gs"
-- [1,3]


intension_vacio:: Contexto Int Int -> Bool
intension_vacio c = intension c [] == atributos c

-- quickCheck intension_vacio
-- OK, passed 100 tests.


extension:: (Eq a, Eq b) => Contexto a b -> [b] -> [a]
extension c ms = [o | o <- objetos c,
                  all (\m -> relacionado c o m) ms]

-- extension ej1 [1]     == "bcdf"
-- extension ej1 [1,2]   == "bcd"
-- extension ej1 [0,1,2] == ""
```

```
-- extension ejd [2,4]
-- "r"


extension_vacio:: Contexto Int Int -> Bool
extension_vacio c = extension c [] == objetos c

-- quickCheck extension_vacio
-- OK, passed 100 tests.


clausuraO :: (Eq a, Eq b) => Contexto a b -> [a] -> [a]
clausuraO c = extension c . intension c


clausuraA :: (Eq a, Eq b) => Contexto a b -> [b] -> [b]
clausuraA c = intension c . extension c


subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]


prop_2_1:: Contexto Int Int -> [Int] -> Property
prop_2_1 c gs =
  (not (null obs) && subconjunto gs obs) ==>
    subconjunto gs (clausuraO c gs)
      where obs = objetos c

-- quickCheck prop_2_1
-- *** Gave up! Passed only 48 tests.

prop_2_1':: Contexto Int Int -> Bool
prop_2_1' c =
    and [subconjunto gs (clausuraO c gs) | gs <- obss c]

-- quickCheckWith (stdArgs {maxSize=6}) prop_2_1'
-- +++ OK, passed 100 tests.
```

```
prop_2_2 :: Contexto Int Int -> Bool
prop_2_2 c =
    and [subconjunto (intension c gs2) (intension c gs1)|
        gs1 <- obss c, gs2 <- extiendeSub gs1 obs]
        where obs = objetos c

extiendeSub ys xs = map (ys++) $ subsequences (xs \\ ys)

-- quickCheckWith (stdArgs {maxSize= 22}) prop_2_2
-- +++ OK, passed 100 tests.
-- (1.55 secs, 1,111,533,312 bytes)


prop_2_3 :: Contexto Int Int -> Bool
prop_2_3 c =
    and [intension c gs == intension c (clausuraO c gs)|
         gs <- obss c]

-- quickCheckWith (stdArgs {maxSize= 22}) prop_2_3
-- +++ OK, passed 100 tests.
-- (0.68 secs, 445,248,392 bytes)


prop_3_1:: Contexto Int Int -> Bool
prop_3_1 c =
    and [subconjunto ms (clausuraA c ms) | ms <- atbs c]

-- quickCheckWith (stdArgs {maxSize= 6}) prop_3_1
-- +++ OK, passed 100 tests.


prop_3_2 :: Contexto Int Int -> Bool
prop_3_2 c =
    and [subconjunto (extension c ms2) (extension c ms1)|
          ms1 <- atbs c, ms2 <- extiendeSub ms1 atb]
        where atb = atributos c
```

```
-- quickCheckWith (stdArgs {maxSize= 22}) prop_3_2
-- +++ OK, passed 100 tests.
-- (9.66 secs, 2367712192 bytes)


prop_3_3 :: Contexto Int Int -> Bool
prop_3_3 c =
    and [extension c ms == extension c (clausuraA c ms)|
            ms <- atbs c]

-- quickCheckWith (stdArgs {maxSize= 6}) prop_3_3
-- +++ OK, passed 100 tests.


prop_4_dcha :: Contexto Int Int -> Bool
prop_4_dcha c =
    and [subconjunto ms (intension c gs) | ms <- atbs c,
           gs <- subsequences (extension c ms)]

-- quickCheckWith (stdArgs {maxSize= 20}) prop_4_dcha
-- +++ OK, passed 100 tests.
-- (0.21 secs, 81,403,264 bytes)


prop_4_izda :: Contexto Int Int -> Bool
prop_4_izda c =
    and [subconjunto gs (extension c ms) | gs <- obss c,
           ms <- subsequences (intension c gs)]

-- quickCheckWith (stdArgs {maxSize= 20}) prop_4_izda
-- +++ OK, passed 100 tests.
-- (0.25 secs, 36749064 bytes)


prodCartesiano:: [a] -> [b] -> [(a,b)]
prodCartesiano xs ys = [(x,y) | x <- xs, y <- ys]
```

```
prop_5_dcha :: Contexto Int Int -> Bool
prop_5_dcha c =
    and [subconjunto (prodCartesiano gs ms) rel |
         gs <- obss c, ms <- subsequences (intension c gs)]
        where rel = relaciones c

-- quickCheckWith (stdArgs {maxSize= 20}) prop_5_dcha
-- +++ OK, passed 100 tests.
-- (0.80 secs, 185002280 bytes)


prop_5_izda :: Contexto Int Int -> Bool
prop_5_izda c =
    and [subconjunto ms (intension c gs) |
         gs <- obss c, ms <- atbs c,
         subconjunto (prodCartesiano gs ms) rel]
        where rel = relaciones c

-- quickCheckWith (stdArgs {maxSize= 20}) prop_5_izda
-- +++ OK, passed 100 tests.
-- (0.94 secs, 174469008 bytes)


unionG :: Eq a => [[a]] -> [a]
unionG = foldl union []

-- unionG [[1,2,3], [3,4,5,1], [9,8,2]]
-- [1,2,3,4,5,9,8]


interseccionG :: Eq a => [[a]] -> [a]
interseccionG = foldl1 intersect

-- interseccionG [[1,2,3], [3,4,5,1], [9,8,2]] == []
-- interseccionG [[1,2,3], [3,4,5,1], [9,8,2,1]] == [1]
```

```
prop_6o :: Contexto Int Int -> Bool
prop_6o c =
     and [intension c (unionG gss) ==
       interseccionG [intension c xs| xs <- gss] | gss<-lss]
         where lss = tail $ subsequences (obss c)

-- quickCheckWith (stdArgs {maxSize= 8}) prop_6o
-- +++ OK, passed 100 tests.
-- (14.66 secs, 3814595432 bytes)


prop_6a :: Contexto Int Int -> Bool
prop_6a c =
     and [intension c (unionG fss) ==
       interseccionG [intension c ys| ys <- fss] | fss<-lss]
         where lss = tail $ subsequences (atbs c)

-- quickCheckWith (stdArgs {maxSize= 10}) prop_6a
-- +++ OK, passed 100 tests.
-- (27.22 secs, 7385014792 bytes)


esConcepto :: (Ord a, Ord b) =>
              Contexto a b -> ([a],[b]) -> Bool
esConcepto c (gs, ms) =
        intension c gs == ms && extension c ms == gs

--   esConcepto ej1 ("abcd",[2]) = True
--   esConcepto ej1 ("",[0]) = False
--   esConcepto ej1 ("ae",[0]) = True


esSubconcepto :: (Ord a, Ord b) =>
        Contexto a b -> ([a],[b]) -> ([a],[b]) -> Bool
esSubconcepto c (gs1,ms1) (gs2,ms2) =
    subconjunto gs1 gs2 && esConcepto c (gs1,ms1) &&
                esConcepto c (gs2,ms2)
```

```
mayorSubconcepto :: (Ord a, Ord b) =>
     Contexto a b -> ([a],[b]) -> ([a],[b]) -> ([a],[b])
mayorSubconcepto c (gs1,ms1) (gs2,ms2) =
    (gs1 `intersect` gs2, clausuraA c (ms1 `union` ms2))


menorSuperconcepto :: (Ord a, Ord b) =>
    Contexto a b -> ([a],[b]) -> ([a],[b]) -> ([a],[b])
menorSuperconcepto c (gs1,ms1) (gs2,ms2) =
    (clausuraO c (gs1 `union` gs2), ms1 `intersect` ms2)


conceptos :: (Ord a, Ord b) => Contexto a b -> [([a],[b])]
conceptos c = [(gs,ms)| gs <- obss c,
               let ms = intension c gs,
               extension c ms == gs]

-- conceptos ejd
-- [("r",[1,3,4,2]),("gr",[1,3,4]),("srp",[1,3,2]),
-- ("gsrp",[1,3]), ("gsrmp",[1])]
-- (0.27 secs, 6506296 bytes)


numConceptos :: (Ord a, Ord b) => Contexto a b -> Int
numConceptos = length . conceptos

-- numConceptos ejd
-- 5


display :: Show a => [a] -> IO()
display [] = return ()
display (x:xs) = do print x
                    display xs
```

```
subc_ref :: Contexto Int Int -> Bool
subc_ref c =
     and [esSubconcepto c (gs1,ms1) (gs1,ms1)|
            (gs1,ms1) <- conceptos c]

-- quickCheckWith (stdArgs {maxSize= 30})  subc_ref
-- +++ OK, passed 100 tests.
-- (1.89 secs, 417540304 bytes)


subc_tran :: Contexto Int Int -> Bool
subc_tran c =
     and [esSubconcepto c (gs1,ms1) (gs3,ms3)|
          (gs1,ms1)<- conceptos c,
          (gs2,ms2)<- conceptos c,
          (gs3,ms3)<- conceptos c,
           esSubconcepto c (gs1,ms1) (gs2,ms2),
           esSubconcepto c (gs2,ms2) (gs3,ms3)]

-- quickCheckWith (stdArgs {maxSize= 20})  subc_tran
-- +++ OK, passed 100 tests.
-- (39.33 secs, 9699472856 bytes)


subc_antisim :: Contexto Int Int -> Bool
subc_antisim c =
     and [(gs1,ms1) == (gs2,ms2)|
          (gs1,ms1)<- conceptos c,
          (gs2,ms2)<- conceptos c,
           esSubconcepto c (gs1,ms1) (gs2,ms2),
           esSubconcepto c (gs2,ms2) (gs1,ms1)]

-- quickCheckWith (stdArgs {maxSize= 20})  subc_antisim
-- +++ OK, passed 100 tests.
-- (1.42 secs, 297539288 bytes)


-- Algoritmos para generar todos los conceptos de un contexto:
```

```
-- Algoritmo 1:

-- Paso 1.

candidatosExtensiones1:: (Ord a, Ord b) =>
                            Contexto a b -> [[a]]
candidatosExtensiones1 c = nub (aux ats [obs])
     where ats = atributos c
           obs = objetos c
           aux [] ac = ac
           aux (y:ys) ac = aux ys (ac ++map (intersect y1) ac)
              where y1 = extension c [y]

-- candidatosExtensiones1 ej1 == ["abcdef","ae","abcd","a",
-- "bcdf","","bcd"]

-- candidatosExtensiones1 ejd
-- ["gsrmp","gsrp","gr","srp","r"]
-- (0.02 secs, 0 bytes)

candidatosExtensiones1_b:: (Ord a, Ord b) =>
                            Contexto a b -> [[a]]
candidatosExtensiones1_b c =
    nub(foldl f [objetos c] (atributos c))
     where f ac y = ac ++ map (intersect (extension c [y])) ac

-- candidatosExtensiones1_b ejd
-- ["gsrmp","gsrp","gr","srp","r"]
-- (0.00 secs, 0 bytes)


-- Paso 2.

candidatosIntensiones1:: (Ord a, Ord b) =>
                            Contexto a b -> [[b]]
candidatosIntensiones1 c =
     map (intension c) (candidatosExtensiones1 c)
```

```
-- candidatosIntensiones1 ejd
-- [[1],[1,3],[1,3,4],[1,3,2],[1,3,4,2]]
-- (0.00 secs, 0 bytes)


-- Paso 3.

generaConceptos1::(Ord a, Ord b) =>
                    Contexto a b -> [([a], [b])]
generaConceptos1 c =
     zip (candidatosExtensiones1 c) (candidatosIntensiones1 c)

-- generaConceptos1 ejd
-- [("gsrmp",[1]),("gsrp",[1,3]),("gr",[1,3,4]),
-- ("srp",[1,3,2]), ("r",[1,3,4,2])]
-- (0.02 secs, 0 bytes)


igualConjunto xs ys = subconjunto xs ys &&
    subconjunto ys xs && length xs == length ys


generaConceptos_prop1:: Contexto Int Int -> Bool
generaConceptos_prop1 c =
     igualConjunto (generaConceptos1 c) (conceptos c)

-- quickCheckWith (stdArgs {maxSize= 20})
-- generaConceptos_prop1
-- +++ OK, passed 100 tests.
-- (0.76 secs, 163404176 bytes)


-- Algoritmo 2:

-- Paso 1.

candidatosIntensiones2:: (Ord a, Ord b) =>
```

```
                              Contexto a b -> [[b]]
candidatosIntensiones2 c = nub (aux obs [ats])
     where ats = atributos c
           obs = objetos c
           aux [] ac = ac
           aux (x:xs) ac = aux xs (ac ++map (intersect x1) ac)
              where x1 = intension c [x]

-- candidatosIntensiones2 ejd
-- [[1,3,4,2],[1,3,4],[1,3,2],[1,3],[1]]
-- (0.02 secs, 0 bytes)


-- Paso 2.

candidatosExtensiones2:: (Ord a, Ord b) =>
                              Contexto a b -> [[a]]
candidatosExtensiones2 c =
     map (extension c) (candidatosIntensiones2 c)

-- candidatosExtensiones2 ejd
-- ["r","gr","srp","gsrp","gsrmp"]
-- (0.00 secs, 0 bytes)


-- Paso 3.

generaConceptos2::(Ord a, Ord b) =>
                    Contexto a b -> [([a], [b])]
generaConceptos2 c =
     zip (candidatosExtensiones2 c) (candidatosIntensiones2 c)

-- generaConceptos2 ejd
-- [("r",[1,3,4,2]),("gr",[1,3,4]),("srp",[1,3,2]),
-- ("gsrp",[1,3]), ("gsrmp",[1])]
-- (0.02 secs, 0 bytes)
```

```
generaConceptos_prop2:: Contexto Int Int -> Bool
generaConceptos_prop2 c =
     igualConjunto (generaConceptos2 c) (conceptos c)

-- quickCheckWith (stdArgs {maxSize= 20})
-- generaConceptos_prop2
-- +++ OK, passed 100 tests.
-- (1.48 secs, 333043352 bytes)


esImplicacion :: (Ord a, Ord b) =>
                    Contexto a b -> ([b],[b]) -> Bool
esImplicacion c (m,n) =
    subconjunto m (atributos c) &&
    subconjunto n (atributos c)


imps :: (Ord a, Ord b) => Contexto a b -> [([b],[b])]
imps c = [(m,n) | m <- atbs c, n <- atbs c]


esModelo :: (Ord a, Ord b) =>
            Contexto a b -> ([b],[b]) -> [b] -> Bool
esModelo c (m,n) ts =
    subconjunto ts (atributos c) && elem (m,n) (imps c) &&
    (not (subconjunto m ts) || subconjunto n ts)


modelosImp :: (Ord a, Ord b) =>
                Contexto a b -> ([b],[b]) -> [[b]]
modelosImp c (m,n) = filter (esModelo c (m,n)) (atbs c)


esModeloConj ::  (Ord a, Ord b) =>
                Contexto a b -> [([b],[b])] -> [b] -> Bool

esModeloConj c ls ts = and [esModelo c (m,n) ts|(m,n) <- ls]
```

```
modelosConjImp :: (Ord a, Ord b) =>
                    Contexto a b -> [([b],[b])] -> [[b]]
modelosConjImp c ls = filter (esModeloConj c ls) (atbs c)


implValidaConj :: (Ord a, Ord b) =>
                    Contexto a b -> ([b],[b]) -> [[b]] -> Bool
implValidaConj c (m,n) tss = all (esModelo c (m,n)) tss


implValida :: (Ord a, Ord b) =>
                Contexto a b -> ([b],[b]) -> Bool
implValida c (m, n) =
    implValidaConj c (m,n) [intension c [o] | o <- objetos c]


impsValidas :: (Ord a, Ord b) => Contexto a b -> [([b],[b])]
impsValidas c = filter (implValida c) (imps c)


prop_10 :: Contexto Int Int -> Bool
prop_10 c = and [subconjunto n (clausuraA c m) |
                    (m,n) <- impsValidas c]

-- quickCheckWith (stdArgs {maxSize= 10}) prop_10
-- +++ OK, passed 100 tests.
-- (0.50 secs, 99461800 bytes)


cerrado :: Eq a => [a] -> [[a]]  -> Bool
cerrado gs uss = gs `elem` uss &&
                and [(interseccionG xs) `elem` uss |
                         xs <- tail (subsequences uss)]
```

```
prop_11_a :: Contexto Int Int  -> Bool
prop_11_a c =
    and [cerrado (atributos c) (modelosConjImp c ls) |
             ls <- subsequences (imps c)]

-- quickCheckWith (stdArgs {maxSize=2}) prop_11_a
-- +++ OK, passed 100 tests.
-- (0.06 secs, 0 bytes)
-- quickCheckWith (stdArgs {maxSize=3}) prop_11_a
-- +++ OK, passed 100 tests.
-- (0.08 secs, 28177144 bytes)


prop_11_b :: Contexto Int Int -> Bool
prop_11_b c = igualConjunto (modelosConjImp c ls)
                            (candidatosIntensiones2 c)
              where ls = impsValidas c

-- quickCheckWith (stdArgs {maxSize=10}) prop_11_b
-- +++ OK, passed 100 tests.
-- (6.15 secs, 1683378216 bytes)


esConsecuencia :: (Ord a, Ord b) =>
    Contexto a b -> ([b],[b]) -> [([b],[b])] -> Bool
esConsecuencia c (m,n) ls = and [esModelo c (m,n) ts |
                        ts <- atbs c, esModeloConj c ls ts]


sonConsecuencia :: (Ord a, Ord b) =>
                     Contexto a b -> [([b],[b])] -> [([b],[b])]
sonConsecuencia c ls = [l | l <-imps c, esConsecuencia c l ls]


adecuadoImpl :: (Ord a, Ord b) =>
                 Contexto a b -> [([b],[b])] -> Bool
adecuadoImpl c ls = and [implValida c l | l <- ls]
```

```
completo :: (Ord a, Ord b) =>
            Contexto a b -> [([b],[b])] -> Bool
completo c ls = and [esConsecuencia c l ls | l<-impsValidas c]


noRedundante :: (Ord a, Ord b) =>
                Contexto a b -> [([b],[b])] -> Bool
noRedundante c ls =
    [l | l <- ls, esConsecuencia c l (delete l ls)] == []


pseudoInt :: (Ord a, Ord b) => Contexto a b -> [b] -> Bool
pseudoInt c [] = [] /= (clausuraA c [])
pseudoInt c p = not (igualConjunto p (clausuraA c p)) &&
                      and [ subconjunto (clausuraA c q) p
                                | q <- init (subsequences p),
                                       pseudoInt c q ]

-- pseudoInt ejd []
-- True
-- pseudoInt ejd [3]
-- False


-- Algoritmo para generar todas las pseudo-intensiones de un
-- contexto:

subconjuntosCard:: [a] -> Int -> [[a]]
subconjuntosCard _ 0      = [[]]
subconjuntosCard [] _     = []
subconjuntosCard (x:xs) k =
    map (x:) (subconjuntosCard xs (k-1)) ++
                            subconjuntosCard xs k
```

```
pseudoIntRes :: (Ord a, Ord b) =>
                Contexto a b -> [b] -> [[b]] -> Bool
pseudoIntRes c [] ss = [] /= (clausuraA c [])
pseudoIntRes c p ss =
    not (igualConjunto p (clausuraA c p)) &&
        and [subconjunto (clausuraA c q) p
                 | q <- cs]
            where cs = intersect ss (init (subsequences p))

pseudoIntensionesCard :: (Ord a, Ord b) =>
                         Contexto a b -> Int -> [[b]]
pseudoIntensionesCard c 0 | [] == (clausuraA c []) = []
                          | otherwise              = [[]]
pseudoIntensionesCard c k =
            union ss [p | p <- cs, pseudoIntRes c p ss]
                where ss = pseudoIntensionesCard c (k-1)
                      cs = subconjuntosCard (atributos c) k


pseudoIntensiones :: (Ord a, Ord b) => Contexto a b -> [[b]]
pseudoIntensiones c = pseudoIntensionesCard c n
    where n = length (atributos c)

-- pseudoIntensiones ejd
-- [[],[1,4],[1,2]]


diferenciaL :: Eq a => [a] -> [a] -> [a]
diferenciaL ys xs = [y | y <- ys, y `notElem` xs]

baseStem :: (Ord a, Ord b) => Contexto a b -> [([b],[b])]
baseStem c = [(p, diferenciaL (clausuraA c p) p)
                | p <- pseudoIntensiones c]

-- baseStem ejd
-- [([],[1]),([1,4],[3]),([1,2],[3])]
```

```
-- ghci> baseStem ej5
-- [([],[1,2,3,4]),([0,1,2,3,4,5],[6,7,8,9,10,11,12]),
-- ([1,2,3,4,6],[5,7,8,9]),([1,2,3,4,7],[5,6,8,9]),
-- ([1,2,3,4,8],[5,6,7,9]),([1,2,3,4,9],[5,6,7,8]),
-- ([1,2,3,4,10],[5,6,7,8,9]),([1,2,3,4,11],
-- [0,5,6,7,8,9,10,12]),([1,2,3,4,12],
-- [0,5,6,7,8,9,10,11])]
-- (35.42 secs, 10429103736 bytes)

-- baseStem ej1
-- [([0,1],[2])]


teorema_1 :: Contexto Int Int -> Bool
teorema_1 c = noRedundante c ls && completo c ls &&
                adecuadoImpl c ls
              where ls = baseStem c

-- quickCheckWith (stdArgs {maxSize=10}) teorema_1
-- +++ OK, passed 100 tests.
-- quickCheckWith (stdArgs {maxSize=30}) teorema_1
-- +++ OK, passed 100 tests.
-- (3.92 secs, 875816224 bytes)


-- ------------------------------------------------------------
```

```
-- GENERADOR DE CONTEXTOS :

transforma:: (Eq a, Eq b) => [(a,b)] -> [(a,[b])]
transforma rs = [(x,img x) | x <- as]
    where as = nub $ map fst rs
          img x = [y | (x', y) <- rs, x' == x]

genContexto :: Gen (Contexto Int Int)
genContexto = do
  xs <- listOf1 arbitrary
  let xs1 = nub $ map abs xs
  ys <- listOf (elements [(x,y) | x <- xs1, y <- xs1])
  return (creaContexto (transforma ys))

instance Arbitrary (Contexto Int Int) where
    arbitrary = genContexto
```