TECNOLÓGICO DE COSTA RICA

Escuela de Computación

Ingeniería en Computación – Lenguajes de Programación (IC-4700)

Asignación #4

Procesamiento simbólico con lenguajes funcionales (Haskell)

Estudiantes:

Sebastián Bermúdez Acuña – 2021110666

Anthony Jiménez Barrantes – 2021022457

Damián Obando Cerdas - 2021047883

Profesor: Ignacio Trejos Zelaya

26 de octubre del 2022

II Semestre, 2022

Índice

Introducción

La programación funcional es un paradigma de la programación, que permite generar un enfoque mayor sobre en el “qué” se está trabajando, en vez del “cómo” se está haciendo, alimenta el uso de recursos como la recursividad y la importancia de orden. El presente trabajo se desarrolla con el propósito de seguir trabajando con el paradigma funcional, en esta ocasión con el lenguaje de programación Haskell, donde se creará y desarrollará diferentes procesadores de expresiones simbólicas, muchas de estas basadas en las desarrolladas previamente en el lenguaje Standard ML, en la asignación 2. Cada implementación y sub-implementación correspondiente será explicada y ejemplificada en todas las posibilidades, para demostrar el correcto funcionamiento de estas, además de darle a entender al lector la forma en que los objetivos fueron logrados. En primer lugar, se les dedicara una parte a las funciones “traducidas” de la asignación 2 a Haskell, luego una demostración y presentación de las funciones que cumplen con la nueva función a hacer, un convertidor de proposiciones lógicas con variables en la forma normal conjuntiva. Además, se demostrará una función que actuará como un simplificador de proposiciones lógicas. Por último, se hablará sobre un análisis de los resultados obtenidos, conclusiones del trabajo, la descripción de las complejidades en el desarrollo, una reflexión sobre el trabajo final y una descripción sobre la división de tareas; correspondientemente.

Función *vars*

Para la función vars se cuenta con un metodo llamado *filter*, el mismo se encarga de filtrar una lista dado una proposición, como segundo método se cuenta con *nub*, el cual se encarga de obtener una lista de duplicados para de esta manera poder eliminarlos. Ambas se utilizan para hacer uso de la funcion *vars*, la cual tiene como objetivo poder obtener todas las variables involucradas dado una proposición, devueltas en una lista de tipo string[1].

La traducción a *Haskell* fue sencilla, la función *filter* fue proporcionada por el profesor y las demás funciones fue usar su equivalente en sintaxis.

Para probar la función vars se necesita primeramente tener el data de tipo proposición, la función *filter* y *nub*, además de la proposición con la que se va a probar la función.

Pruebas

Prueba con una variable de tipo proposición que solamente tiene una variable llamada “p”, el resultado obtenido es una lista de string con solo el elemento “p”.

```
ghci> pru00 = Conjuncion(p,Negacion(p))
ghci> pru00
Conjuncion (Variable "p",Negacion (Variable "p"))
ghci> vars pru00
["p"]
ghci>
```

Prueba con una variable de tipo proposición que tiene 2 variables (“p” y “q”), el resultado obtenido será una lista de string con ambas variables.

```
ghci> pru6 = Implicacion(Conjuncion(p,q),Disyuncion(q,p))
ghci> pru6
Implicacion (Conjuncion (Variable "p",Variable "q"),Disyuncion (Variable "q",Variable "p"))
ghci> vars pru6
["p","q"]
```

Prueba con una variable de tipo proposición que tiene 3 variables (“p”, “q”, “r”), el resultado obtenido será una lista de string con las 3 variables.

```
ghci> pru01 = Disyuncion(p, Conjuncion(Negacion(q),r))
ghci> pru01
Disyuncion (Variable "p",Conjuncion (Negacion (Variable "q"),Variable "r"))
ghci> vars pru01
["p","q","r"]
```

Función gen_bools

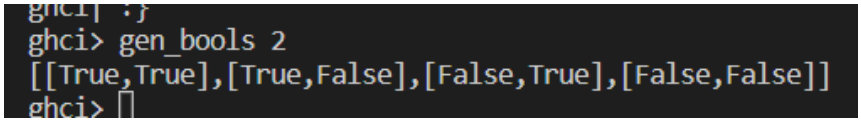
La función *gen_bools* se encarga de recibir un numero *n* que indicará la cantidad de variables de la proposición para así mapear en una matriz de 2^*n* arreglos de valores booleanos que representará cada una de sus filas, retorna en una lista de valores booleanos. Utiliza la función *cons* que construye listas "*curryficadas*" y la función de *map* que mapea sobre una lista.

Su traducción a *Haskell* fue bastante sencilla, ya que solo hubo pocos cambios con respecto a su implementación en *Standard ML*.

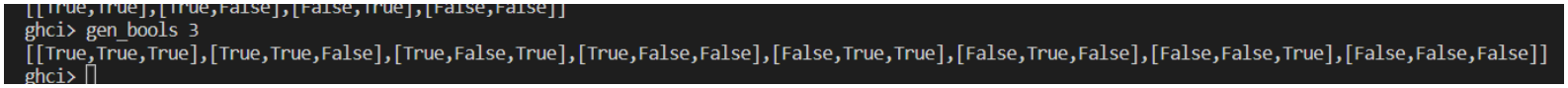
Para probar la función se necesita solamente indicar el número de variables.

Pruebas

Prueba pasando como parámetro el número entero 2, lo que debería generar 4 arreglos booleanos



Prueba pasando como parámetro 3, lo que debería de generar 8 arreglos booleanos



Función as_vals

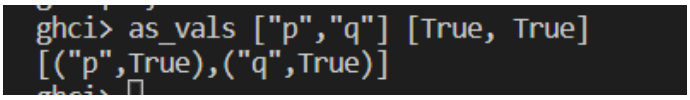
La función *as_vals* se encarga de recibir 2 listas, una de variables y otra de valores booleanos, produciendo una lista de pares ordenados (string, bool). Utiliza la función *zip*, que dado 2 listas las combina y produce una lista de pares ordenados, si estas no son del mismo tamaño devolverá una lista vacía, ya que es la versión permisiva.

Su implementación en *Haskell* no tuvo casi cambios con respecto a su versión en *Standard ML*.

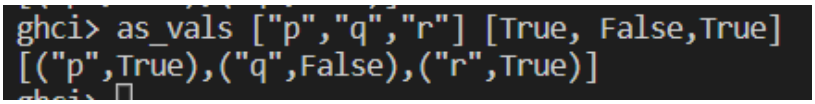
Para probar la función *as_vals* se necesita una lista de variables junto con la lista de booleanos.

Pruebas

Se prueba *as_vals* con una lista de 2 variables, “p” y “q” con los valores true en ambos casos



Prueba con una lista de 3 variables, “p”, “q” y “r” con los valores true, false, true respectivamente



Función evalProp

La funcion *evalProp* se encarga de evaluar proposiciones, recibe por parámetro un ambiente y la proposición a evaluar, puede realizar varias operaciones como lo son la negación, conjunción, disyunción, implicación y equivalencia. La función retorna un valor booleano.

Su implementación en *Haskell* no tuvo casi cambios con respecto a su versión en *Standard ML*, excepto que no hace uso de ambientes e identificadores y solamente da una excepción de tipo "error."

Para probar la función evalProp se debe dar el ambiente o contexto a evaluar y la proposición.

Pruebas

La función es capaz de probar una proposición sin variables, esto es posible si se da un ambiente vacío, se probará con una implicación de *false* a *true*, que retornará *true* como resultado.

```
ghci> prop3
Implicacion (Constante False,Constante True)
ghci> evalProp [] prop3
True
```

Como la función es capaz de evaluar las filas individualmente como una tabla de verdad, se probará una implicación de "p" a "q", la variable "p" será *true* y la variable "q" será *false*, lo que retornará *false* como resultado.

```
ghci> pru1 = Implicacion(p,q)
ghci> lista = [("p",True),("q",False)]
ghci> evalProp lista pru1
False
```

Función tabla_dato

La función *tabla_dato* recibe una proposición e imprime la tabla de verdad en forma de dato, esta recorrerá cada posible combinación booleana para guardar su resultado en una lista, todas las combinaciones se guardarán de la forma variable+booleano, luego se mostrarán al usuario.

Su implementación en *Haskell* fue muy sencilla ya que casi no tuvo cambios ni en sintaxis ni en lógica.

Para probar la función solo hace falta la proposición a evaluar y las funciones anteriores.

Código de la función:

El código de la función *tabla_dato* se encuentra en la subcarpeta llamada “Código Fuente”, en el archivo tabla_dato.hs

Pruebas:

Se probará con una proposición que es una disyunción de 2 variables

```
Disyuncion (Variable "p",Variable "q")
ghci> tabla_dato prop1
[([("p",True),("q",True)],True),([("p",True),("q",False)],True),([("p",False),("q",True)],True),([("p",False),("q",False)],False)]
ghci>
```

Se probará con una implicación de 2 variables

```
ghci> pru1
Implicacion (Variable "p",Variable "q")
ghci> tabla_dato pru1
[([("p",True),("q",True)],True),([("p",True),("q",False)],False),([("p",False),("q",True)],True),([("p",False),("q",False)],True)]
ghci>
```

Se probará con una conjunción de 3 variables, todas estarán negadas.

```
ghci> prueba
Conjuncion (Conjuncion (Negacion (Variable "p"),Negacion (Variable "q")),Negacion (Variable "r"))
ghci> tabla_dato prueba
[([("p",True),("q",True),("r",True)],False),([("p",True),("q",True),("r",False)],False),([("p",True),("q",False),("r",True)],False),([("p",True),("q",False),("r",False)]
,False),([("p",False),("q",True),("r",True)],False),([("p",False),("q",True),("r",False)],False),([("p",False),("q",False),("r",True)],False),([("p",False),("q",False),(
"r",False)],True)]
```

Función *taut*

La función *taut* recibe una proposición y determinará si se trata de una tautología, esta recorrerá cada evaluación de la proposición, si encuentra que alguna de las evaluaciones es de valor booleano *false*, esta se detendrá ya que no se trataría de una tautología. La función también es capaz de evaluar sin variables. Utiliza todas las funciones anteriores, como puede ser *vars*, *gen_bools*, entre otras.

Su implementación en *Haskell* no pudo ser tan completa ya que no hace uso de excepciones a diferencia de su implementación en *Standard ML*¸ si cumple su función ya que puede identificar correctamente las tautologías, pero no puede dar la razón exacta del por qué no es una tautología.

Para probar la función solo hace falta la proposición a evaluar y las funciones anteriores.

Código de la función:
El código de la función *taut* se encuentra en la subcarpeta llamada “Código Fuente”, en el archivo taut.hs

Pruebas

Se probará con una proposición que no es una tautología.

```
ghci> pru7
Implicacion (Disyuncion (Variable "q",Variable "p"),Conjuncion (Variable "p",Variable "q"))
ghci> taut pru7
"*** Exception: No es tautologia
CallStack (from HasCallStack):
```

Prueba con una tautología, por lo tanto, dará que sí es una tautología.

```
ghci> pru6 = Implicacion(Conjuncion(p,q),Disyuncion(q,p))
ghci> taut pru6
"implicacion (conjuncion (variable p, variable q), disyuncion (variable q, variable p)) Si es una tautologia"
```

Se probará con una contradicción, todas sus evaluaciones son falsas.

```
ghci> pru8
Disyuncion (Conjuncion (Negacion (Constante False),Constante False),Conjuncion (Constante True,Negacion (Constante True)))
ghci> taut pru8
"*** Exception: No es tautologia
```

Función *fnd*

Para poder crear la función *fnd*, primero se entendió el funcionamiento de la función normal disyuntiva, al crear una tabla de verdad sobre una proposición se debe tomar todas aquellas evaluaciones de la proposición donde sea verdadero, unir las variables mediante conjunciones y entre las evaluaciones con disyunciones. Primeramente, se sacaron la cantidad de variables involucradas en la proposición, ya que si es 0 significa que no contiene variables y solamente se evaluará en un ambiente vacío, si no es 0, gracias a que la función evalúa cada fila de la tabla de verdad, se tomará en cuenta solamente cuando la evaluación es verdadera. Si es verdadera se insertará en una lista que contendrá todas las evaluaciones verdaderas.

Como la cantidad de variables no es 0 se recorrerán todas las combinaciones posibles de la tabla de verdad y se guardarán en la lista llamada asociación, se recorrerá esta lista con una función llamada *recorrerListaAsociación*, si la lista a recorrer está vacía significa que se trata de una contradicción y por lo tanto es falso, de lo contrario usara de la función *unirProp*, esta tiene como objetivo unir mediante conjunciones las diferentes proposiciones, luego de unirlas se unirá con una disyunción a no ser que sea el último elemento de la lista, donde no ocurre.

Su implementación es *Haskell* fue sencilla ya que no hubo casi cambios, la lógica fue la misma.

Código de la función:

El código de la función *fnd* se encuentra en la subcarpeta llamada “Código Fuente”, en el archivo fnd.hs

Descripción de funciones auxiliares

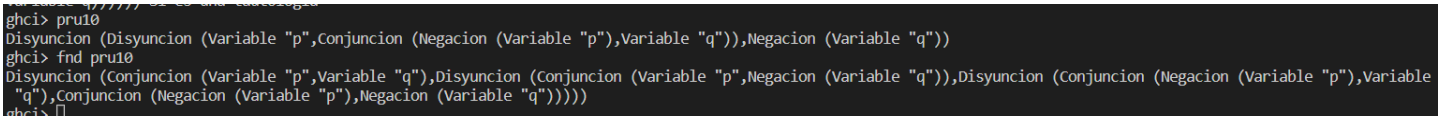
unirProp: Une las variables junto con su valor booleano mediante conjunciones, la función *fnd* lo utiliza para crear la forma normal disyuntiva.

recorrer: Recorre cada fila de la tabla de verdad creada, mediante las combinaciones booleanas creadas.

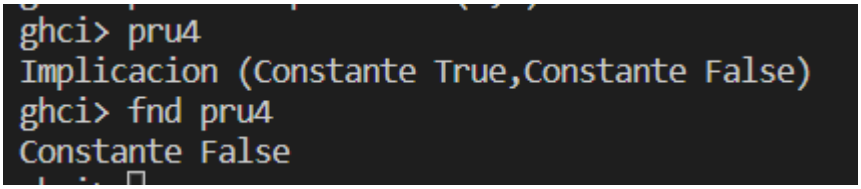
recorrerListaAsociacion: Recorre la lista de evaluaciones que hayan dado valores verdades a la hora de evaluarla, si la lista enviada está vacía quiere decir que es una contradicción y dará que es falso, si no, hará uso de la función *unirProp* para crear la forma normal disyuntiva.

Pruebas

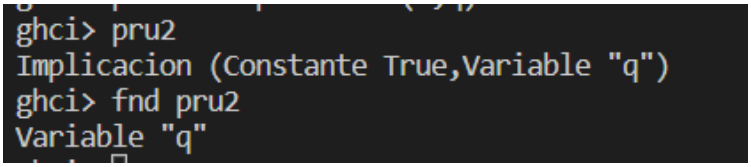
La primera prueba será con una proposición normal, dará como resultado una proposición



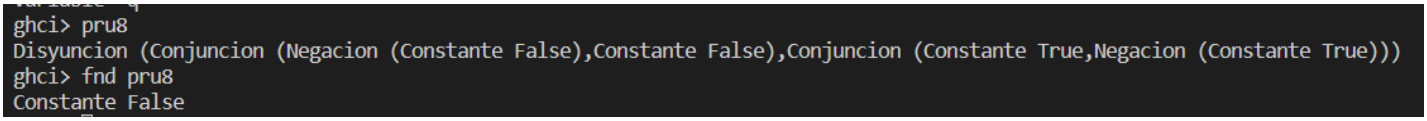
Se probará con una proposición que solamente tiene constantes, dará como resultado false.



Se probará con una proposición conformada por una constante pero solamente con una variable, dará como resultado solamente la variable “q”.



Se probará con una contradicción, dará como resultado false.



Función *fnc*

Para desarrollar la función se tomó como base el de la Función *fnd,* ya que su funcionamiento es muy parecido y gracias a que la Función *fnd* estaba ya traducida en *Haskell* mucho del trabajo ya se encontraba realizado. Las funciones esenciales usadas no se modificaron, si no solamente el orden de algunas acciones o la proposición usada. El primer cambio ocurre en el *if* que verifica si el resultado de la fila es verdadero o falso, como la función normal conjuntiva requiere que sea falso se cambió el orden de las acciones para poner en prioridad cuando la asociación de falso. El segundo cambio ocurre en la función *recorrerListaAsociacion,* en este ya no se unen mediante Disyunciones si no Conjunciones. El ultimo cambio ocurre en la función *unirProp*, las Conjunciones se reemplazan por disyunciones y el orden a evaluar si una variable es negativa o no también cambia.

Código de la función:

El código de la función *fnc* se encuentra en la subcarpeta llamada “Código Fuente”, en el archivo fnc.hs

Descripción de funciones auxiliares

unirProp: Une las variables junto con su valor booleano mediante conjunciones, la función fnc lo utiliza para crear la forma normal conjuntiva.

recorrer: Recorre cada fila de la tabla de verdad creada, mediante las combinaciones booleanas creadas.

recorrerListaAsociacion: Recorre la lista de evaluaciones que hayan dado valores falsos a la hora de evaluarla, si la lista enviada está vacía quiere decir que es una tautología y dará que es verdadero, si no, hará uso de la función unirProp para crear la forma normal conjuntiva.

Pruebas:

La primera prueba se realizará con una tautología, como resultado dará *true*, ya que no contiene evaluaciones falsas y por lo tanto forma normal conjuntiva no existe.

```
ghci> pru10
Disyuncion (Disyuncion (Variable "p",Conjuncion (Negacion (Variable "p"),Variable "q")),Negacion (Variable "q"))
ghci> fnc pru10
Constante True
```

Se probará con una proposición que solamente tiene constantes, dará como resultado false.

```
ghci> pru4
Implicacion (Constante True,Constante False)
ghci> fnc pru4
Constante False
```

Se probará con una proposición conformada por una constante pero solamente con una variable, dará como resultado solamente la variable “q”.

```
ghci> pru2
Implicacion (Constante True,Variable "q")
ghci> fnc pru2
Variable "q"
```

La última prueba será sacar la forma normal conjugada de una proposición que involucra 3 variables, por esta razón es bastante extensa.

```
ghci> pru01
Disyuncion (Variable "p",Conjuncion (Negacion (Variable "q"),Variable "r"))
ghci> fnc pru01
Conjuncion (Disyuncion (Variable "p",Disyuncion (Negacion (Variable "q"),Negacion (Variable "r"))),Conjuncion (Disyuncion (Variable "p",Disyuncion (Negacion (
Variable "q"),Variable "r")),Disyuncion (Variable "p",Disyuncion (Variable "q",Variable "r"))))
```

Función *simpl*

Se utilizaron las reglas de simplificación dadas por el profesor en el archivo de la asignación 4, que son equivales de [Murillo, 2010] y [Morgan, 1994].

Para diseñar cada una de las reglas de simplificación propuestas, primero se entendió la lógica detrás de cada una de estas. Al ya entender la lógica se procedió con el desarrollo de la función. Se hizo uso sobre todo del uso del operador "case _ of", este operador fue sumamente útil, ya que así podíamos verificar todos los casos posibles a evaluar, haciendo no solo que se pudiera implementar exitosamente, si no también que el código quedara limpio y agradable a la vista para cualquier programador. Se hizo uso de funciones externas para el apoyo de la función principal, así como de librerías externas.

Código de la función:

El código de la función *simpl* se encuentra en la subcarpeta llamada "Código Fuente", en el archivo simpl.hs

Descripción de las funciones auxiliares:

mostCommonElem: Función que se encarga de encontrar el elemento más común en una lista.

borrarElemento: Borra un elemento especifico de una lista.

elementosRestantes: Obtiene todos los elementos de una lista a la vez que se elimina el que contiene más apariciones.

Pruebas:

Se mostrarán las pruebas de las 10 primeras reglas de simplificación, para poder observar las restantes reglas dirigirse a **Apéndices.**

```
ghci> regla1
Implicacion (Variable "p",Variable "q")
ghci> simpl regla1
Disyuncion (Negacion (Variable "p"),Variable "q")
ghci> regla2
Negacion (Negacion (Variable "p"))
ghci> simpl regla2
Variable "p"
ghci> regla3
Negacion (Disyuncion (Variable "p",Variable "q"))
ghci> simpl regla3
Conjuncion (Negacion (Variable "p"),Negacion (Variable "q"))
ghci> regla4
Negacion (Conjuncion (Variable "p",Variable "q"))
ghci> simpl regla4
Disyuncion (Negacion (Variable "p"),Negacion (Variable "q"))
ghci> regla5
Conjuncion (Disyuncion (Variable "p",Variable "q"),Disyuncion (Variable "p",Variable "r"))
ghci> simpl regla5
Disyuncion (Variable "p",Conjuncion (Variable "q",Variable "r"))
ghci> regla6
Conjuncion (Disyuncion (Variable "p",Variable "q"),Disyuncion (Variable "r",Variable "p"))
ghci> simpl regla6
Disyuncion (Variable "p",Conjuncion (Variable "q",Variable "r"))
ghci> regla7
Conjuncion (Disyuncion (Variable "q",Variable "p"),Disyuncion (Variable "p",Variable "r"))
ghci> simpl regla7
Disyuncion (Variable "p",Conjuncion (Variable "q",Variable "r"))
ghci> regla8
Conjuncion (Disyuncion (Variable "q",Variable "p"),Disyuncion (Variable "r",Variable "p"))
ghci> simpl regla8
Disyuncion (Variable "p",Conjuncion (Variable "q",Variable "r"))
ghci> regla9
Disyuncion (Conjuncion (Variable "p",Variable "q"),Conjuncion (Variable "p",Variable "r"))
ghci> simpl regla9
Conjuncion (Variable "p",Disyuncion (Variable "q",Variable "r"))
ghci> regla10
Disyuncion (Conjuncion (Variable "p",Variable "q"),Conjuncion (Variable "r",Variable "p"))
ghci> simpl regla10
Conjuncion (Variable "p",Disyuncion (Variable "q",Variable "r"))
ghci>
```

Análisis de los resultados obtenidos:

Como se pudo observar la función hace perfectamente cada una de las reglas de simplificación, estas tienen el resultado esperado sin ninguna sorpresa, todos de acuerdo con la tabla proporcionada por el profesor Trejos Zelaya.

Discusión y Análisis de Resultados

Después de concluir con el desarrollo de todas las funciones propuestas, es válido decir que se cumplió exitosamente lo esperado. Gracias a las experiencias pasadas con el lenguaje de programación *Standard ML* fue muy sencillo hacer el cambio a *Haskell*. El paradigma de programación funcional se pudo entender a un nuevo nivel de profundidad gracias a la nueva experimentación. También se aprendió la traducción del lenguaje de *Standard ML* a *Haskell*, estos 2 lenguajes siendo muy parecidos hizo que sea una experiencia interesante al ver las pequeñas diferencias en cada uno. Las funciones de la asignación 2 dadas por el profesor Trejos Zelaya fueron traducidas correctamente, incluyendo la función *fnc* desarrollada por nuestra parte. Por su parte la nueva función *fnc* cumple con la estandarización de la forma normal conjuntiva de la lógica booleana. La función *simpl* hace satisfactoriamente las reglas de simplificación de la 1 a la 45. Cada una está documentada, hubo distintas pruebas y se considera satisfecha su implementación, aunque como siempre, todo trabajo puede llegar a mejorarse.

Conclusiones

Siempre es importante estar en constante cambio. Quedarse estancado no es bueno y siempre hay que probar de todo. Esto también aplica para los lenguajes de programación, hay que tener un poco de conocimiento en cada aspecto de los lenguajes más importantes. En este caso que nos concierne en referente al lenguaje de programación *Haskell*, aunque ya se tuviera experiencia al trabajar en la asignación 2 con un lenguaje muy similar. Sus sutiles diferencias son importantes. Se genero un nuevo conocimiento en el aspecto tanto de la programación funcional como del uso de proposiciones lógicas booleanas. Usando sus reglas de simplificación y entendiendo el uso de estas. Tras la finalización del trabajo se presentan buenas sensaciones finales. Y se agradece el desafío de enfrentarnos a retos descocidos para ponernos a prueba en el ámbito de la programación.

Problemas y limitaciones

Uno de los problemas encontrados fue con respecto a trabajar con el lenguaje *Haskell*, a pesar de ser dentro de un paradigma conocido, siempre el desafío de aprender algo nuevo puede intimidar en un principio. Esto se logró resolver al investigar sobre su sintaxis, funcionamiento, instalamiento y haciendo pruebas sobre programas.

Al diseñar la función *fnc* no hubo muchos problemas, se tomó de base la función *fnd* que fue traducida correctamente y solamente se adoptó a las reglas de la función normal conjuntiva. El único problema encontrado fue que ninguno de los integrantes se acordaba al 100% de la función normal conjuntiva. Entonces se procedió a hacer un repaso grupal para poder entenderla.

Con respecto a la función *simpl* si hubieron mucho más problemas, se tuvo que hacer varias pruebas para encontrar maneras efectivas de obtener las variables en juego y poder hacer comparaciones, para saber que regla usar en cada caso.

Reflexión

El uso del lenguaje de programación *Haskell y Standard ML* presento un reto divertido a la vez que interesante, ya que, aunque fuera un nuevo lenguaje y pareciera un reto difícil a primera vista. Gracias a las clases y sobre todo al haber trabajado anteriormente con programación funcional no fue tan complicado como pudo haber parecido en un principio. Los ejemplos proporcionados por el profesor Trejos Zelaya, así como las clases fueron de gran ayuda a la hora de programar en este nuevo lenguaje. En términos generales. Se aprendió mucho más sobre la programación funcional, sobre un lenguaje extremadamente importante tanto en el pasado como hoy en día. Y sobre todo provechosa.

Descripción y División de Tareas

- Sebastián Bermúdez Acuña (2021110666): Encargado del informe técnico (formato, escritura, orden), implementación de la mitad de las reglas de simplificación.
- Anthony Jiménez Barrantes (2021022457): Traducción de todas las funciones de *Standard ML* a *Haskell*, implementación de la función *fnc*.
- Damián Obando Cerdas (2021047883): Implementación de la otra mitad de las reglas de simplificación.

Referencias Bibliográficas

Trejos, I. (06 de noviembre de 2022). Asignación_4_2022-2_-_recursos_de_apoyo. San José, Costa Rica.

Making Our Own Types and Typeclasses - Learn You a Haskell for Great Good! (s. f.). http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Haskell Code by HsColour. (s. f.). https://downloads.haskell.org/%7Eghc/6.12.1/docs/html/libraries/base-4.2.0.0/src/Data-List.html

[Morgan, 1994] Morgan, Carroll. Programming from specifications, 2nd ed. Prentice Hall International, 1994. Ver: Appendix A: Some laws for predicate calculation, particularmente la sección A.1

[Murillo, 2010] Murillo Tsijli, Manuel. Introducción a la Matemática Discreta, 4ta ed. Editorial Tecnológica de Costa Rica, 2010. Ver: secciones 1.3 Leyes de la lógica y 1.5 Formas normales.

Apéndices

Instrucciones para ejecutar el programa

Para poder ejecutar el programa se debe primero tener el código fuente, este se encuentra en la carpeta llamada “Código Fuente”. Se debe tener alguna versión de *Haskell* instalada en el computador, a preferencia del usuario. Para poder probar todas las funcionalidades del programa se deben compilar los archivos en un orden especifico. Primeramente, se debe compilar todo lo encontrado en el archivo “sintax.hs”, luego se compila lo encontrado en el archivo “vars.hs”, se continua con compilar lo encontrado en “gen_bools.hs”, se continua con “as_vals.hs”, se continua con “evalProp.hs”, se continua con “taut.hs”, “tabla_dato.hs”, “fnc.hs”, “fnd.hs” y por último “simpl.hs”. Con esto se podrá ejecutar el programa correctamente.

Descripción general de las pruebas diseñadas para validar su programa

Se usaron pruebas utilizadas en la asignación 2, así se pudo verificar mejor que las funciones estén correctas al observar que dieron los mismos resultados que en *Standard ML*, las pruebas para la función *simpl* fueron copiar exactamente todas las pruebas básicas de las reglas de simplicación. Todas estas pruebas se encuentran en “sintax.hs”.

Pruebas de función *simpl*.

Se probarán las reglas restantes de la 10 a la 45.

Pruebas de la regla 11 a la 20

```
ghci> regla11
Disyuncion (Conjuncion (Variable "q",Variable "p"),Conjuncion (Variable "p",Variable "r"))
ghci> simpl regla11
Conjuncion (Variable "p",Disyuncion (Variable "q",Variable "r"))
ghci> regla12
Disyuncion (Conjuncion (Variable "q",Variable "p"),Conjuncion (Variable "r",Variable "p"))
ghci> simpl regla12
Conjuncion (Variable "p",Disyuncion (Variable "q",Variable "r"))
ghci> regla13
Conjuncion (Variable "p",Variable "p")
ghci> simpl regla13
Variable "p"
ghci> regla14
Disyuncion (Variable "p",Variable "p")
ghci> simpl regla14
Variable "p"
ghci> regla15
Disyuncion (Variable "p",Constante False)
ghci> simpl regla15
Variable "p"
ghci> regla16
Disyuncion (Constante False,Variable "p")
ghci> simpl regla16
Constante False
ghci> regla17
Conjuncion (Variable "p",Constante True)
ghci> simpl regla17
Variable "p"
ghci> regla18
Conjuncion (Constante True,Variable "p")
ghci> simpl regla18
Constante True
ghci> regla19
Negacion (Constante False)
ghci> simpl regla19
Constante True
ghci> regla20
Negacion (Constante True)
ghci> simpl regla20
Constante False
ghci>
```

Pruebas de la regla 21 a la 30

```
ghci> regla21
Disyuncion (Variable "p",Negacion (Variable "p"))
ghci> simpl regla21
Constante True
ghci> regla22
Disyuncion (Negacion (Variable "p"),Variable "p")
ghci> simpl regla22
Constante True
ghci> regla23
Conjuncion (Variable "p",Negacion (Variable "p"))
ghci> simpl regla23
Constante False
ghci> regla24
Conjuncion (Negacion (Variable "p"),Variable "p")
ghci> simpl regla24
Constante False
ghci> regla25
Conjuncion (Variable "p",Constante False)
ghci> simpl regla25
Constante False
ghci> regla26
Conjuncion (Constante False,Variable "p")
ghci> simpl regla26
Constante False
ghci> regla27
Disyuncion (Variable "p",Constante True)
ghci> simpl regla27
Constante True
ghci> regla28
Disyuncion (Constante True,Variable "p")
ghci> simpl regla28
Constante True
ghci> regla29
Disyuncion (Variable "p",Conjuncion (Variable "p",Variable "q"))
ghci> simpl regla29
Variable "p"
ghci> regla30
Disyuncion (Conjuncion (Variable "p",Variable "q"),Variable "p")
ghci> simpl regla30
Variable "p"
```

Pruebas de la regla 31 a la 40

```
ghci> regla31
Disyuncion (Variable "p",Conjuncion (Variable "q",Variable "p"))
ghci> simpl regla31
Variable "p"
ghci> regla32
Disyuncion (Conjuncion (Variable "q",Variable "p"),Variable "p")
ghci> simpl regla32
Variable "p"
ghci> regla33
Conjuncion (Variable "p",Disyuncion (Variable "p",Variable "q"))
ghci> simpl regla33
Variable "p"
ghci> regla34
Conjuncion (Disyuncion (Variable "p",Variable "q"),Variable "p")
ghci> simpl regla34
Variable "p"
ghci> regla35
Conjuncion (Variable "p",Disyuncion (Variable "q",Variable "p"))
ghci> simpl regla35
Variable "p"
ghci> regla36
Conjuncion (Disyuncion (Variable "q",Variable "p"),Variable "p")
ghci> simpl regla36
Variable "p"
ghci> regla37
Disyuncion (Variable "p",Conjuncion (Negacion (Variable "p"),Variable "q"))
ghci> simpl regla37
Disyuncion (Variable "p",Variable "q")
ghci> regla38
Disyuncion (Variable "p",Conjuncion (Variable "q",Negacion (Variable "p")))
ghci> simpl regla38
Disyuncion (Variable "p",Variable "q")
ghci> regla39
Disyuncion (Conjuncion (Negacion (Variable "p"),Variable "q"),Variable "p")
ghci> simpl regla39
Disyuncion (Variable "p",Variable "q")
ghci> regla40
Disyuncion (Conjuncion (Variable "q",Negacion (Variable "p")),Variable "p")
ghci> simpl regla40
Disyuncion (Variable "p",Variable "q")
```

Pruebas de la regla 41 a la 45

```
ghci> regla41
Conjuncion (Variable "p",Disyuncion (Negacion (Variable "p"),Variable "q"))
ghci> simpl regla41
Conjuncion (Variable "p",Variable "q")
ghci> regla42
Conjuncion (Variable "p",Disyuncion (Variable "q",Negacion (Variable "p")))
ghci> simpl regla42
Conjuncion (Variable "p",Variable "q")
ghci> regla43
Conjuncion (Disyuncion (Negacion (Variable "p"),Variable "q"),Variable "p")
ghci> simpl regla43
Conjuncion (Variable "p",Variable "q")
ghci> regla44
Conjuncion (Disyuncion (Variable "q",Negacion (Variable "p")),Variable "p")
ghci> simpl regla44
Conjuncion (Variable "p",Variable "q")
ghci> regla45
Implicacion (Variable "p",Implicacion (Variable "q",Variable "r"))
ghci> simpl regla45
Implicacion (Disyuncion (Variable "p",Variable "q"),Variable "r")
```