Asignación #3: Procesamiento simbólico con lenguajes funcionales

Estudiantes:

Marco Antonio Reveiz Rojas 2019053583

Steven Granados Acuña 2019037999

Profesor:

Ignacio Trejos Zelaya

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

IC-4700 Lenguajes de Programación

Introducción

El objetivo del desarrollo de este trabajo es realizar tres funciones con ayuda de archivos anteriormente programados por el profesor. Cada función cumple una tarea distinta. La función fnd se basa en convertidor de proposiciones lógicas con variables hacia la forma normal disyuntiva. La función bonita funciona como un “impresor” de las mismas. Por último, la función simpl desarrolla una simplificación de proposiciones lógicas con el uso de diez leyes de la lógica. Este trabajo se divide en tres partes designadas a las tres funciones de las cuales se hablaron anteriormente. La primera sección se caracteriza por describir el proceso de programación y planteamiento de la función fnd. La segunda sección está compuesta por la misma estrategia de planteamiento pero de la función bonita, y por último la tercera está compuesta por la función simpl.

Funciones que se desarrollaron:

Función Fnd

Esta función consistía en diseñar un conversor de proposiciones lógicas en su forma normal disyuntiva. Al principio, no se sabía cómo implementar una función que permitiera realizar la conversión principalmente por un obstáculo el cual consiste en la no comprensión de la forma en que se programa en el lenguaje Standard ML. Por esto, se solicitó una consulta al profesor, en esta reunión se lograron aclarar las cuestiones antes mencionadas y se logró plantear una estrategia que resultó ser exitosa.

Estrategia y funciones auxiliares

La función fnd se compone de varias funciones auxiliares que realizan ciertas tareas de composición y descomposición del parámetro recibido, el cual es un datatype brindado por el profesor llamado Proposición. La conversión que realiza fnd se compone varias etapas, pero se podrían determinar tres etapas distintas. La primera es realizada por las funciones previamente brindadas por el profesor y funciones de Standard ML que dan datos generales. En esta etapa se utilizan las funciones de vars que retorna una lista de las variables de la proposición recibida como parámetro. Luego, se genera la cantidad de booleanos de acuerdo a una función dada previamente llamada gen_bools con la cantidad de variables de acuerdo a la función de SML length. La segunda etapa se compone de las cuatro funciones auxiliares:

1. **FndAux**: La primera función auxiliar la cual recorre los datos realizados por la etapa anterior, se llama función fndAux y sirve de manera recursiva retornando una lista de tuplas con el nombre de la variable y su valor (verdadero o falso). Está tiene como propósito, determinar cuál son los casos verdaderos en de la tabla de verdad de la proposición. Esta función utiliza dos funciones dadas por el profesor, evalProp que evalúa un caso junto con su proposición y as_vals que recibe la lista de booleanos junto con la lista de las variables de dicha proposición recibida ambas previamente calculadas en la primera etapa. De as_vals se obtiene una variable llamada asociación que es el caso a evaluar, esta variable sirve como parámetro en evalProp donde se calcula si este caso es verdadero o no de acuerdo a la proposición dada. Si este caso es verdadero se agrega a una lista que se va creando recursivamente, y llama de nuevo a la función eliminando la cabeza de la lista recibida y realizando el mismo proceso con la cola.
2. **toProp**: Una vez fndAux retorna la lista con los casos verdaderos se deben ir transformando estos casos en proposiciones. El objetivo de toProp, es convertir los casos dados por fndAux en forma de lista tuplas a una lista de listas pero con cada sublista compuesta de las variables con su valor de verdad (si es false se crea la variable negada, si no se crea sin negar) ya dadas como una Proposición. Esta función es recursiva. Y retorna una lista de sublistas de variables.
3. **gen_cony**: Al crear la lista de toProp, se deben unir las variables de los casos evaluados en conjunciones por lo que en propósito de gen_cony es transformar la lista de sublistas de variables en una lista de sublistas de conjunciones por medio de hacer un map de gen_cony. Una vez se ingresa la lista el map se encarga de crear una conjunción para cada sublista y las variables que se encuentren dentro de ella de manera recursiva.

4. **gen_disy**: Por último, la forma normal disyuntiva los casos verdaderos se unen por medio de una disyunción por lo que con la lista que se retorna en el map de gen_cony, se crea una sola proposición unida por disyunciones que es creada por gen_disy. Una vez se crea esta proposición se obtiene de manera oficial la forma normal disyuntiva concluyendo el proceso de fnd.

La tercera y última etapa de fnd consiste en "acomodar los datos" con manejo de variables y las funciones auxiliares de acuerdo a temporales. Una vez se realizan estas operaciones se retorna como variable la forma normal disyuntiva de la proposición dada.

Pruebas y resultados obtenidos

El experimento requiere que para demostrar el funcionamiento óptimo de la función se hagan pruebas. En el caso de fnd, implementamos un total de tres pruebas distintas de proposiciones que fueron obtenidas de tablas de verdad que se buscaron en fuentes de internet.

Lista de Pruebas:

```
(*PRUEBA de FND*)
val fndp1 =  ~:(variable ("p"):||:variable ("q")):=>:(variable ("p"):=>:variable ("r"))
val fndp2 =  (variable ("p") :&&: variable ("r")) :<=>: (variable ("q") :||: variable ("r"))
val fndp3 =  ((variable ("p") :&&: variable ("r")) :||: ( variable ("q") :&&: ~: (variable ("r"))))
```

Las anteriores pruebas están basadas en las siguientes tablas de verdad

Prueba 1:

p	q	r	$(p \vee q)$	$\neg(p \vee q)$	$(p \to r)$	$\neg(p \vee q) \to (p \to r)$
V	V	V	V	F	V	V
V	V	F	V	F	F	V
V	F	V	V	F	V	V
V	F	F	V	F	F	V
F	V	V	V	F	V	V
F	V	F	V	F	V	V
F	F	V	F	V	V	V
F	F	F	F	V	V	V

Prueba 2:

p	q	r	$(p \wedge r)$	$(q \vee r)$	$(p \wedge r) \leftrightarrow (q \vee r)$
V	V	V	V	V	V
V	V	F	F	V	F
V	F	V	V	V	V
V	F	F	F	F	V
F	V	V	F	V	F
F	V	F	F	V	F
F	F	V	F	V	F
F	F	F	F	F	V

Prueba 3:

p	q	r	p ^ q	¬r	q ^ ¬r	(p ^ q) v (q ^ ¬r)
V	V	V	V	F	F	V
V	V	F	V	V	V	V
V	F	V	F	F	F	F
V	F	F	F	V	F	F
F	V	V	F	F	F	F
F	V	F	F	V	V	V
F	F	V	F	F	F	F
F	F	F	F	V	F	F

Después de ejecutar fnd con las anteriores pruebas se obtuvieron los siguientes resultados:

Resultado 1:

```
fnd fndp1;
> val it =
    disyuncion(conjuncion(variable "p",
                          conjuncion(variable "q", variable "r")),
               disyuncion(conjuncion(variable "p",
                                     conjuncion(variable "q",
                                                negacion(variable "r"))),
                          disyuncion(conjuncion(variable "p",
                                                conjuncion(negacion(variable "q"),
                                                           variable "r")),
                                     disyuncion(conjuncion(variable "p",
                                                           conjuncion(negacion(variable "q"),
                                                                      negacion(variable "r"))),
                                                disyuncion(conjuncion(negacion(variable "p"),
                                                                      conjuncion(variable "q",
                                                                                 variable "r")),
                                                           disyuncion(conjuncion(negacion(variable "p"),
                                                                                 conjuncion(variable "q",
                                                                                            negacion(variable "r"))),
                                                                      disyuncion(conjuncion(negacion(variable "p"),
                                                                                            conjuncion(negacion(variable "q"),
                                                                                                       variable "r")),
                                                                                 conjuncion(negacion(variable "p"),
                                                                                            conjuncion(negacion(variable "q"),
                                                                                                       negacion(variable "r"))))))))))
    : Proposicion
```

Resultado 2:

```
fnd fndp2;
> val it =
    disyuncion(conjuncion(variable "p",
                          conjuncion(variable "r", variable "q")),
               disyuncion(conjuncion(variable "p",
                                     conjuncion(variable "r",
                                                negacion(variable "q"))),
                          disyuncion(conjuncion(variable "p",
                                                conjuncion(negacion(variable "r"),
                                                           negacion(variable "q"))),
                                     conjuncion(negacion(variable "p"),
                                                conjuncion(negacion(variable "r"),
                                                           negacion(variable "q"))))))
    : Proposicion
```

Resultado 3:

```
fnd fndp3;
> val it =
    disyuncion(conjuncion(variable "p",
                          conjuncion(variable "r", variable "q")),
               disyuncion(conjuncion(variable "p",
                                     conjuncion(variable "r",
                                                negacion(variable "q"))),
                          disyuncion(conjuncion(variable "p",
                                                conjuncion(negacion(variable "r"),
                                                           variable "q")),
                                     conjuncion(negacion(variable "p"),
                                                conjuncion(negacion(variable "r"),
                                                           variable "q"))))) :
  Proposicion
```

Referencias

Monografias.com - tesis, documentos, publicaciones y recursos Educativos. (2021, June 14). Monografías. https://www.monografias.com/

Murillo, M. M. (2010). *Introducción a la matemática discreta* (4th ed.) [E-book]. https://manfredohurtado.jimdofree.com/app/download/11633309877/Introducci%C3%B3n+a+la+Matem%C3%A1tica+Discreta+4ed+%28+PDFDrive.com+%29.pdf?t=1583289219

Función bonita

La función bonita se encarga de realizar una “impresión” de una proposición dada de manera que se pueda observar de manera clara con los símbolos adecuados. Esta función no cumple con funciones auxiliares y funciona de manera recursiva.

Estrategia y funciones auxiliares

Se utilizó como base la función imprimir que brindó el profesor en el archivo sintax.sml. Esta función no fue complicada debido a que su propósito no es difícil de entender y es bastante simple su programación. Se planteó una función que se maneja a través de casos los cuales en caso de ser una proposición compuesta se vuelve a llamar así misma y realiza el mismo proceso. En caso de no serlo se imprime normal sin llamarse de nuevo.

Pruebas y resultados obtenidos

A continuación se presenta una imagen con las pruebas diseñadas para bonita:

```
(*PRUEBA PARA BONITA*)
val bonita1 = (p :||: q) :=>: (p :&&: t)
val bonita2 = f :<=>: ~:(p :&&: q)
```

Los resultados obtenidos son los siguientes:

Prueba 1:

```
bonita bonita1;
> val it = "((verbatim(p) \\/ verbatim(q)) => (verbatim(p) /\\ true))" : string
```

Prueba 2:

```
bonita bonita2;
> val it = "(false <=> ~((verbatim(p) /\\ verbatim(q))))" : string
```

Función simpl

La función de simpl, se basa en realizar la simplificación de una proposición brindada como parámetro con una serie de leyes lógicas seleccionadas de acuerdo al criterio de los programadores. Al ser un trabajo realizado por dos personas se implementaron diez leyes.

Estrategia y funciones auxiliares

Al empezar la programación de simpl no se comprendía la forma en la cual debía plantearse la estrategia. Esto hizo que se acudiera a una consulta con el profesor, ahí fue donde se aclaró que debía construirse como una función de casos recursiva. Primero debían determinarse cuáles leyes de lógica se iban a utilizar. Para esto se escogieron las siguientes: Ley de inversos, Ley de neutros, Ley de dominación, idempotencia, doble negación e implicación y disyunción. Al ser una función recursiva se tomaron varios casos base como el hecho de que la proposición fuera una sola variable o una sola constante con su valor de verdad. De esta manera, una vez no se pudiera simplificar más con estas leyes se retorna la proposición sola. No se requirieron funciones auxiliares debido a que la función se llama a sí misma de forma recursiva y aplica la misma las mismas evaluaciones a cada una de las sub proposiciones que componen una proposición más grande.

Pruebas y resultados obtenidos

Para esta función se diseñaron 3 pruebas que utilizan las leyes de lógica implementadas en simpl, esto por que en los libros siempre aparecen simplificaciones que usan leyes como asociatividad, distributividad, etc. Y por estas simplificaciones intermedias simpl no es capaz de llegar a la simplificación máxima de esas proposiciones encontradas en los libros.

Las pruebas diseñadas y sus respectivas soluciones son las siguientes:

Prueba 1:

$(P \wedge P) \Rightarrow (Q \vee \neg Q)$	
$(P) \Rightarrow (Q \vee \neg Q)$	Idempotencia
$P \Rightarrow (V_0)$	Inverso
$\neg P \vee V_0$	ID
V_0	Dominación

Prueba 2:

$\neg(\neg P \vee F_0)$	
$\neg\neg P$	Neutro
P	Doble negación

Prueba 3:

$(\neg P \vee F_0) \Rightarrow (Q \wedge V_0)$	
$(\neg P) \Rightarrow (Q \wedge V_0)$	Neutro
$\neg P \Rightarrow (Q)$	Neutro
$P \vee Q$	ID

Las pruebas escritas en código fuente de SML son las siguientes:

```
val p = variable "p";
val q = variable "q";
val z = variable "z";
val f = constante false;
val t = constante true;

(* PRUEBA PARA SIMPL*)
val hitotsu = (p :&&: p) :=>: (q :||: (~:q))
val futatsu = ~:(~:p :||: f)
val mittsu = ((~:(p) :||: f) :=>: (q :&&: t))
```

Los resultados de ejecutar simpl con las pruebas anteriores son los siguientes:

```
simpl hitotsu;
> val it = constante true : Proposicion
-


simpl futatsu;
> val it = variable "p" : Proposicion
-



simpl mittsu;
> val it = disyuncion(variable "p", variable "q") : Proposicion
```

Referencias

Murillo, M. M. (2010). *Introducción a la matemática discreta* (4th ed.) [E-book]. https://manfredohurtado.jimdofree.com/app/download/11633309877/Introducci%C3%B3n+a+la+Matem%C3%A1tica+Discreta+4ed+%28+PDFDrive.com+%29.pdf?t=1583289219

Análisis de Resultados

Función fnd

Calificación	3	2	1	Comentario
3	Obtiene la FND de la proposición brindada de forma correcta	Obtiene la FND de la proposición brindada de forma parcial	No obtiene la FND de la proposición brindad	Realiza la tarea principal sin ninguna complicación
3	Valida todos los casos base como proposiciones “vacías”, etc.	Valida algunos casos base como proposiciones “vacías”, etc.	No valida los casos base como proposiciones “vacías”, etc.	

Función bonita

Calificación	3	2	1	Comentario
3	Imprime correctamente la proposición	Imprime parcialmente la proposición	No Imprime correctamente la proposición	Cumple correctamente con esta tarea
2	Imprime con un formato agradable	Imprime con un formato parcialmente agradable	No Imprime con un formato agradable	Algunos símbolos como “\ /” no se pueden imprimir de esa forma exacta porque SML no lo permite
1	Elimina todos los paréntesis innecesarios	Elimina parcialmente todos los paréntesis innecesarios	No elimina los paréntesis innecesarios	No conocemos cómo se procesan los string en SML por lo que no sabemos cómo detectar paréntesis innecesarios

Función simpl

Calificación	3	2	1	Comentario
2	Simplifica al máximo la proposición brindada	Simplifica parcialmente la proposición brindada	No simplifica la proposición brindada	En algunas pruebas notamos que el algoritmo no es capaz de hacer recursión sobre la proposición y devuelve el mismo valor de entrada
3	Es capaz de utilizar todas las reglas implementadas correctamente	Es capaz de utilizar casi todas las reglas implementadas correctamente	No es capaz de utilizar todas las reglas implementadas correctamente	Usa las reglas cuando el caso hace match con el patrón
2	La función toma en cuenta todos los posibles casos	La función toma en cuenta casi todos los posibles casos	La función toma en cuenta sólo casos base	El compilador envía un warning indicando que no todos los casos están tomados en cuenta

Problemas Encontrados

Una dificultad que se nos presentó al inicio fue a la hora buscar información sobre el lenguaje de programación Standard ML. La información que se puede encontrar al buscar no es tan amplia, o es muy complicada de comprender debido a que no viene de una manera simple y entendible, además los ejemplos que circulan por la web no son tan variados. Por esto, al empezar a realizar las funciones se tuvieron varios problemas para comprender el sintaxis del lenguaje, por lo que dificulto el proceso inicial. Pero con aclaraciones del profesor mediante una sesión de consulta y los ejemplos brindados en los archivos que nos facilitó se logró comprender en cierta medida la manera en la que funciona el lenguaje. A parte de problemas de sintaxis, se podría considerar complicado el instalar el compilador pero en el caso de los participantes del proyecto, no fue así. Se utilizaron dos compiladores populares Moscow ML y New Jersey ML. Pero a la hora de compilar y ejecutar los archivos no hubo problema alguno. La función simpl fue sin dudas la más difícil de implementar ya que requiere de contemplar muchos casos que a menos de que se pruebe y se obtengan errores no son fáciles de detectar. Para bonita, al no tener un conocimiento tan profundo de cómo funcionan los string en SML hizo imposible realizarle mejoras para que imprima únicamente los paréntesis necesarios

Conclusión

Trabajar con el paradigma funcional y Standard ML fue una experiencia muy enriquecedora pero a la vez muy desafiante. Al estar tan acostumbrados al paradigma imperativo y orientado a objetos se debe hacer un cambio radical en la forma de pensar para resolver los problemas que se presentan en el paradigma funcional. Se notó la falta de estructuras que puedan guardar resultados temporales o que representen objetos que puedan guardar información que facilite la resolución de problemas ya que en el paradigma funcional solo se trabaja con resultados de funciones anteriores, además, el bajo nivel que representa SML con respecto a lenguajes como Python o Java hace que la programación se termine asemejando a ASM. Una cosa muy importante que hay que recalcar es que las soluciones en su mayoría están diseñadas de manera recursiva, por lo que aumenta la complejidad de los algoritmos a desarrollar, pero se consigue a su vez formas muy rápidas de solucionar problemas que en otros paradigmas o lenguajes consumirían bastantes recursos, por ejemplo con bucles iterativos. En general el dominio del lenguaje SML tiene una curva bastante empinada ya que la documentación es escasa y tosca para la lectura y comprensión, además de que hay bastantes limitaciones a la hora de trabajar con el paradigma funcional, por lo que SML es muy poderoso en su área pero no es “todo terreno” como lo podría ser Python.

Referencias

Murillo, M. M. (2010). *Introducción a la matemática discreta* (4th ed.) [E-book]. https://manfredohurtado.jimdofree.com/app/download/11633309877/Introducci%C3%B3n+a+la+Matem%C3%A1tica+Discreta+4ed+%28+PDFDrive.com+%29.pdf?t=1583289219

Monografias.com - tesis, documentos, publicaciones y recursos Educativos. (2021, June 14). Monografías. https://www.monografias.com/