Universidad ORT Uruguay
Facultad de Ingeniería

$\mathcal{Z}$orro: Constructor de Demostraciones Calculacionales

Entregado como requisito para la obtención del título de Licenciado en Sistemas

Juan Michelini - 163408

Tutor: Álvaro Tasistro

2013

Yo, Juan Michelini, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Abstract

Al investigar un sistema de demostración formal, es habitual concentrarse primero en las propiedades lógicas del mismo, haciendo énfasis en la completitud, la consistencia y la decidibilidad.

Al enseñar un sistema de demostración formal es habitual concentrarse en el proceso de formalización y en la correctitud de cada manipulación.

Esto resulta en que las demostraciones formales adquieran fama de tediosas en exceso y carentes de elegancia. Sin embargo, nada está más lejos de nuestra experiencia. Consideramos que las demostraciones formales pueden lograr un grado de elegancia inalcanzable por otros métodos. Pero para llegar a este fin, se requiere prestar atención constante a cómo se construye la demostración.

Un enfoque distinto es el de la matemática calculacional de Dijkstra, que prioriza la demostración formal elegante sobre cualquier otra consideración. En esta tesis formalizaremos un método de construcción de demostraciones formales inspirado en la matemática calculacional de Dijkstra. El método se formalizará como un programa de Haskell.

Palabras clave

Lógica; Matemática calculacional; Dijkstra; Demostración automática de teoremas

Índice

1 Motivación

En los últimos años, en simultáneo con el surgimiento de la ciencia de la computación, se han hecho grandes avances en el campo de la formalización y manipulación de sistemas formales.

Uno de estos grandes avances ha sido la formalización de las fórmulas lógicas y de las demostraciones compuestas por las mismas.

La formalización de demostraciones permite decir exactamente cuando una demostración es correcta y cuándo no.

Naturalmente surge la siguiente ambición, formalizar la construcción de demostraciones. Este campo ha tenido mucho crecimiento bajo el nombre de demostradores automáticos de teoremas o asistentes de demostración. Los primeros buscan construir programas que demuestren sin intervención humana, los segundos asistir al matemático en su trabajo.

Notamos que falta una tercera área, aquella que busca formalizar cómo trabaja el matemático, no con el fin de asistirlo ni de automatizar su trabajo, sino con el fin de alcanzar un mayor nivel de comprensión sobre la actividad matemática.

Sin embargo, investigadores como E. W. Dijkstra, David Gries y Netty van Gasteren han hecho grandes avances en esta dirección bajo lo que ellos denominan el método calculacional.

El método calculacional solo admite aquellas demostraciones formales que se componen mediantes reescrituras de una fórmula en otra que preserve su valor. Esta limitación se usa como base para estudiar en detalle qué reescrituras son útiles y cuáles son evitables. Siendo consciente de la metodología con la que se calcula (se seleccionan las reescrituras), se aumenta la probabilidad de llegar a una demostración lo suficientemente corta para que sea considerara elegante. El método calculacional considera una demostración elegante, sólo cundo ha evitado toda manipulación formal que se pueda evitar.[**?**]

Esta tesis surge del interés de empezar a formalizar el método calcula-

cional.

A pesar de los humildes objetivos de esta tesis, nuestros objetivos a largo plazo son mucho más ambiciosos. Deseamos simplificar el argumento matemático, desmitificar el proceso de invención matemática y transformar el diseño de demostraciones elegantes en una disciplina enseñable. [1]

1.1 Objetivo

El objetivo es formalizar un método de construcción de demostraciones formales inspirado en la matemática calculacional. Para esta primera incursión, decidimos limitarnos a la lógica proposicional.

Dado que la matemática calculacional busca primordialmente la elegancia, la formalización deberá construir demostraciones elegantes.

La formalización se efectuará en Haskell, lo que resultará en un programa. El programa podría ser considera como un demostrador automático de teoremas, pero este nombre resulta engañoso. Nos interesa más la forma de construcción de la demostración que la demostración misma. Es por esto que preferimos llamarlo constructor de demostraciones calculacionales[1].

El nombre completo del programa es Zorro: Constructor de Demostraciones Calculacionales. En la jerga de Dijkstra, se llama "conejo" a aquellos pasos en una demostración que se hacen sin justificación aparente. Dijkstra considera que realizar un paso sin justificación es el equivalente a sacar un conejo de la galera. Se cumple con el objetivo, pero no se sabe como [2] .[2]

Decidimos llamarle Zorro al programa no solo porque es un animal conocido por su astucia, sino porque además es un animal que caza conejos.

1.1.1 Como herramienta educativa

Zorro construye demostraciones sumamente detalladas de forma muy similar a como las construiría un matemático que aplicara le metodología de Dijkstra.

[1] Otro nombre apropiado podría ser compositor de demostraciones calculacionales. Dado que no solo busca crear una demostración, sino que busca crear una elegante. Sin embargo consideramos que el nombre es demasiado pretencioso para esta humilde primera versión y lo reservamos para futuras versiones.

[2] Consideramos que un traducción al español más apropiada que "conejo" sería "galerazo".

Esto lo convierte en una herramienta educativa útil para aquellos cursos donde se está introduciendo la lógica formal por primera vez.

El alumno podría estudiar las demostraciones producidas por Zorro para adquirir una comprensión mayor de qué manipulaciones realizar en cada momento.

1.1.2 Como un asistente de demostración

Zorro sirve como asistente de demostración. El matemático puede hacer uso de Zorro para encontrar simplificaciones a ciertas fórmulas en vez de realizarlas a mano o para buscar una equivalencia que pueda resultar útil como lema en otras demostraciones.

1.1.3 Como un vehículo para formalizar heurísticas

Zorro es un humilde comienzo en la formalización de un método calculacional más general. Nuestra intención es continuar desarrollándolo en otras instancias para, en algún momento, llegar a una formalización del método calculacional en lógica de predicados y luego en un álgebra abstracta.

Zorro ha servido para mapear el espacio de dificultades que pueden surgir en un proyecto más ambicioso.

2 Formato de demostraciones

En matemática calculacional las demostraciones son listas de fórmulas intercaladas con comentarios.[3] Por ejemplo si se quiere demostrar que $\neg\neg\neg(\alpha\wedge\beta)$ es equivalente a $(\neg\alpha \vee \neg\beta)$

$\neg\neg\neg(\alpha \wedge \beta)$
$=$ {doble negación}
$\neg(\alpha \wedge \beta)$
$=$ {de Morgan}
$(\neg\alpha \vee \neg\beta)$

La demostración más sencilla, es demostrar que dos fórmulas sintácticamente iguales son equivalentes:

α
$=$ {*identidad*}
α

Pero dado que es redundante, preferiremos decir solamente:

α

Si permitimos la demostración de una única fórmula, podemos formalizar el formato de demostración de la siguiente manera:

```
1 data Proof a = Line a | Link a (Comment) (Proof a)
```

Una vez formalizado el formato, podemos programar operaciones sencillas que nos devuelvan la primera y la última fórmula en la demostración. O operaciones más complicadas como la que remueve los pasos redundantes:

```
1 firstF (Line x) = x
2 firstF (Link x _ _) = x
3
```

```
4 lastF (Line x) = x
5 lastF (Link x s xs) = (lastF xs)
6
7 removeRedundantSteps::Eq a=>Proof a->Proof a
8 removeRedundantSteps (Line x) = (Line x)
9
10 removeRedundantSteps (Link x s (Line y))
11            | x == y = (Line y)
12            | otherwise = (Link x s (Line y))
13
14 removeRedundantSteps (Link x s (Link y s' ys))
15            | x == y = removeRedundantSteps(Link y s' ys)
16            | otherwise = (Link x s (removeRedundantSteps ((Link y s
                ' ys))))
```

Estas operaciones resultarán ser de gran utilidad más adelante.

Volviendo a la primera demostración de que $\neg\neg\neg(\alpha \wedge \beta)$ es equivalente a $(\neg\alpha \vee \neg\beta)$. Resulta interesante derivar la demostración en sentido inverso. Es decir, partiendo de $(\neg\alpha \vee \neg\beta)$ demostrar $\neg\neg\neg(\alpha \wedge \beta)$.

$(\neg\alpha \vee \neg\beta)$
$=$ {de Morgan}
$\neg(\alpha \wedge \beta)$
$=$ {doble negación}
$\neg\neg\neg(\alpha \wedge \beta)$

Pero esta última demostración resulta más difícil de leer. Resulta natural deshacernos de una doble negación, pero no resulta natural introducirla. Para facilitar la lectura de las demostraciones, vamos a introducir en los comentarios un símbolo que indique que dirección es la más sencilla de leer.

Formalizamos los comentarios de la siguiente manera:

```
1 data Sign = Up | Down
2 data Comment = Com Sign String
```

Haciendo uso de estos símbolos, resulta más simple de leer.

$(\neg\alpha \vee \neg\beta)$
$=$ {$\uparrow$ de Morgan}
$\neg(\alpha \wedge \beta)$

$= \{\uparrow \text{doble negación}\}$
$\neg\neg\neg(\alpha \wedge \beta)$

3 Lógica proposicional

Por lógica proposicional entendemos la lógica cuyas fórmulas sólo permiten representar proposiciones. Por ejemplo, si interpretamos p como "llueve" y q como "hay tormenta", interpretaremos la fórmulas $p \wedge q$ como "llueve y hay tormenta". Donde el valor de la fórmula $p \wedge q$ depende de el valor de p y de q.

La lógica proposicional es decidible. Es decir, existen algoritmos que siempre pueden determinar si dos fórmulas cualesquiera son equivalentes.

Esta es una propiedad interesante ya que, una vez diseñado el algoritmo, no se requiere razonamiento para determinar si dos fórmulas son equivalentes. Sin embargo, al no requerir razonamiento, no logran revelar nada más allá del resultado. Lo cual es el interés principal en esta tesis.

También es importante notar que no todos los cálculos son decidibles, por lo que cualquier método que asuma la decidibilidad presentaría problemas a la hora de generalizarlo a otros cálculos (la lógica de predicados).

Es por esto que vamos desentendernos de la decidibilidad de la lógica proposicional para concentrarnos en formalizar el método de razonamiento matemático calculacional. Aún cuando este evade la formalización y muchas veces no llega a un resultado concreto, por que depende fuertemente en la experiencia y juicio del matemático.

3.1 Conectivas

Haremos uso de las variables proposicionales (p, q, r....) para denotar valores de verdad. Los valores de verdad los denotaremos con $\top$ y $\bot$. El primero denota el valor verdadero y se lee "top", el segundo denota el valor de verdad falso y se lee "bottom".

La lógica proposicional tradicionalmente hace uso de las siguientes conectivas binarias: conjunción ($\wedge$), disyunción ($\vee$), implicación ($\rightarrow$), doble implicación ($\leftrightarrow$). También hace uso de la conectivas unaria: negación ($\neg$).

Este conjunto de conectivas es el usado por convención. Pero no necesariamente el más conveniente. Las conectivas $\vee$ y $\wedge$ cumplen las propiedades de

asociación, conmutatividad y absorción. Esto significa que ni los paréntesis ni el orden ni la cantidad de veces que aparece una subexpresión en una expresión del estilo $p \vee ((q \vee p) \vee r)$ importa. Por lo que puede ser simplificada en $p \vee q \vee r$.

Además de que mantienen una simetría entre sí, dictada por de Morgan $\neg p \vee \neg q = \neg(p \wedge q)$ y $\neg p \wedge \neg q = \neg(p \vee q)$.

Estas propiedades hacen que el $\vee$ y $\wedge$ sean muy convenientes.

Con la implicación $\rightarrow$, nos llevamos una sorpresa; no es ni asociativa, ni conmutativa. Su uso debe a que surge naturalmente de la forma de razonar basada en causa y efecto. Como la forma de razonar que buscamos formalizar es ecuacional e intenta ignorar las interpretaciones causales, vamos a ignorar el $\rightarrow$. En cambio, cada vez que querramos escribir $p \rightarrow q$, escribiremos $\neg p \vee q$ que es una expresión equivalente pero por hacer uso del $\vee$ es mucho más atractiva ecuacionalmente.

La decisión de evitar el $\rightarrow$ trae grandes ventajas. Un ejemplo es la equivalencia $p \rightarrow (q \rightarrow r) = q \rightarrow (p \rightarrow r)$ que expresada como disyunción no es más que una aplicación de la conmutatividad $\neg p \vee \neg q \vee r = \neg q \vee \neg p \vee r$. Al evitarlo, también se evita cargar con los conceptos de "pruebas directas", "pruebas indirectas" y del "contrapositivo".[4][5]

¿Qué podemos esperar de la doble implicación $\leftrightarrow$? Como su nombre indica es conmutativa, sin embargo ahí acaban las virtudes de su nombre. La fórmula $p \leftrightarrow q$, debe ser leída como "p si y solo si q" lo cual presenta grandes dilemas gramaticales.[6] Además, se vuelve muy tentador leerlo como "si p entonces q y si q entonces p", lo cual hace uso de la implicación que ya decidimos evitar.

Si prestamos atención a lo que hace la conectiva, notamos que $p \leftrightarrow q$ solo devuelve verdadero cuando $p = q$. Lo cual es mucho más atractivo para razonar calculacionalmente. Sin embargo, estamos usando $=$ solo con valores de verdad, lo que lo diferencia del $=$ general que puede comparar objetos cualesquiera. Para asentuar esta diferencia haremos uso del símbolo $\equiv$ y leeremos $p \equiv q$ como p equivale a q.

El nuevo nombre sigue sugiriendo que la conectiva es conmutativa. Es fácil probar que $\equiv$ es asociativo. Estas dos propiedades indican que no importa el orden ni los paréntesis si concatenamos varias fórmulas con $\equiv$. Desgraciadamente $\equiv$ no cumple la propiedad de absorción. Es decir que $p \equiv p \neq p$. Por lo que si concatenamos la misma fórmula varias veces con $\equiv$ importa la cantidad de veces que lo hagamos.

Ahondando un poco más, notamos que $p \equiv p = \top$ y $\top \equiv p = p$. Haciendo uso de estas dos propiedades es fácil probar por inducción que en fórmula de la forma $p \equiv p \equiv ...p \equiv p$ sera equivalente a p si la cantidad de veces que aparece p es impar y sera equivalente a $\top$ sino.[7]

Consideramos conveniente el uso de $\equiv$ y de hecho la consideramos una conectiva que simplificaría muchos argumentos matemáticos si fuera usada más.[8]

Sobre la negación $\neg$ no tenemos mayores observaciones.

3.2 Representación de fórmulas

El método para construir pruebas calculaciones fue diseñado con lápiz y papel en mente. Este medio fuerza cierta forma sobre el método. En particular, fuerza la forma visible del método. Cómo parte de la investigación consideramos varias formas de representar las fórmulas, buscando determinar cuales fueron las influencias positivas del lápiz y papel sobre el método y cuales fueron negativas.

Una forma fue representar las fórmulas como su árbol sintáctico. Pero esto forzaba grandes problemas a la hora de decidir cual era el árbol sintáctico más conveniente. Es decir, dado que $\wedge$ es asociativo. El árbol sintáctico de $p \wedge q \wedge r$, puede coincidir con el árbol sintáctico de $(p \wedge q) \wedge r$ o el de $p \wedge (q \wedge r)$. Una decisión que el matemático pospone hasta el ultimo momento, considerando ambos arboles sintácticos idénticos.

Por otro lado probamos representar $p \wedge q \wedge r$, como un árbol con $\wedge$ como

raíz y un conjunto de hijos (p, q, r). Esta representación por medio de conjuntos, garantiza la asociatividad[1], la conmutatividad y la absorción. Lo cual lo hace muy atractivo para representar $\wedge$ y $\vee$, pero no $\equiv$ (ya que carece de absorción). Esto nos disuade del uso de conjuntos, ya que nos gustaría representar $\wedge$, $\vee$ y $\equiv$ de la manera más similar posible.

Otro problema de la representación con conjuntos es que al no respetar el orden, la prueba construida se vuelve difícil de leer en expresiones grandes.

Finalmente nos decidimos por representar $p \wedge q \wedge r$ como una raíz $\wedge$ con una serie de hijos ordenados. Esto significa que si nos importa el orden y la cantidad, pero ignoramos en algunos casos los paréntesis. Es decir, permitimos representar todo lo representable con lápiz y papel. Sin embargo, aprendimos que ciertas estructuras deben ser consideradas iguales aún cuando tienen arboles distintos. Por ejemplo $p \vee q$ y $q \vee p$. Lidiaremos con esto más adelante.

3.2.1 Formalización de la representación

Llamamos FL al lenguaje que vamos a considerar:

```
1 data FL = Top | Bot | Var Variable | N FL | C[FL] | D[FL] | E[FL]
2 --Decimos Xa en vez de a, porque en Haskell los valores inician con una
      mayúscula
3 data Variable = Xa|Xb|Xc|Xd|Xe|Xf|Xg|Xh|Xi|Xj|Xk|Xl|Xm|Xn|Xo|Xp|Xq|
      Xr|Xs|Xt|Xu|Xv|Xw|Xx|Xy|Xz
```

N es la negación, C la conjunción, D la disyunción y E la equivalencia. Por ejemplo:

$\neg(p \vee q \vee (q \equiv r))$ se escribiría N(D[Var Xp,Var Xq,E[Var Xq, Var Xr]])

[1]Garantiza la asociatividad siempre y cuando no permitamos que dentro del conjunto haya otro árbol con raíz $\wedge$.

Notese que aunque permitimos ignorar los paréntesis para representar directamente $p \wedge q \wedge r$ también permitimos introducirlos para representar $p \wedge (q \wedge r)$.

3.3 Navegación

Al construir una prueba, se parte de cierta expresión y se busca llegar a otra expresión mediante una serie de sustituciones permitidas. A la expresión original la llamaremos origen, a la expresión objetivo la llamaremos destino y a la serie de substituciones permitida la llamaremos repertorio. El matemático hace uso constante de estos tres elementos; eligiendo en cada paso una substitución del repertorio con la esperanza de que lo aproxime al destino. Esta decisión esta llena de incertidumbre, ya que la gran mayoría de substituciones nos distanciaran del destino. Denominaremos al criterio para seleccionar la substitución adecuada, la estrategia.

Para nosotros entonces, el método de construcción de demostraciones calculacionales se reducirá en estos 4 elementos: el origen, el destino, el repertorio y la estrategia.

Haremos uso de la siguiente notación para referirnos que queremos pasar de una expresión origen o a una expresión destino d, haciendo uso de un repertorio r y una estrategia e:

$$o \underset{r}{\overset{e}{\mathcal{Z}}} d$$

Dado que la gran mayoría de veces el repertorio y la estrategia se mantendrán constantes y únicos, también permitiremos la abreviación:

$$o \mathcal{Z} d$$

3.4 Repertorio

3.5 Equivalencias permitidas

Permitiremos cuando cumplan las propiedades de conmutatividad, asociatividad, absorción y neutro.

También permitiremos la doble negación, de Morgan, y la distribución de la negación sobre la equivalencia. $(\neg(\alpha \equiv \beta) = (\neg\alpha \equiv \beta))$

Permitiremos la distribución de la conjunción sobre la disyunción, la distribución de la disyunción sobre la conjunción. Por último, permitiremos las definiciones de la equivalencia en términos de conjunción y disyunción. $p \equiv q = (\neg p \vee q) \wedge (p \vee \neg q)$ y $p \equiv q = (p \wedge q) \vee (\neg p \vee \neg q)$

3.6 Generalizaciones

Admitiremos, donde sea posible, las generalizaciones de las fórmulas anteriormente mencionadas. Por ejemplo, admitiremos la generalización de de Morgan en ambos sentidos.

$$\neg(\alpha_1 \vee \alpha_2 \vee ... \vee \alpha_n) = (\neg\alpha_1 \wedge \neg\alpha_2 \wedge ... \wedge \neg\alpha_n)$$

$$\neg(\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n) = (\neg\alpha_1 \vee \neg\alpha_2 \vee ... \vee \neg\alpha_n)$$

3.7 Distinción entre reescritura y equivalencia

Estudiando las equivalencias permitidas, vemos que cada una puede ser leída en dos direcciones. Hacemos uso de esta distinción ya que transformar $\neg\neg\alpha$ en α es algo casi siempre deseable y que no requiere mayor justificación mientras que transformar α en $\neg\neg\alpha$ es raramente deseable sin una razón explicita.

Llamaremos reescritura a cada lectura de una equivalencia.

Por ejemplo la equivalencia $\neg\neg\alpha = \alpha$, puede ser leída como la reescritura $\neg\neg\alpha \Longrightarrow \alpha$ y la reescritura $\alpha \Longrightarrow \neg\neg\alpha$.

Por $\alpha \Longrightarrow \beta$ entendemos la reescritura de α en términos de β.

4 Estrategia

Es responsabilidad de la estrategia coordinar el uso de las reescrituras con el fin de transformar el origen en el destino. Pero no es su única responsabilidad, también le exigimos que las coordine para lograr una demostración calculacional y que justifique la motivación detrás de cada reescritura.

¿Qué motivaciones permitiremos entonces? La motivación por excelencia es que dos expresiones son sintácticamente idénticas y por lo tanto no se requiere hacer ninguna reescritura, pero esta rara vez requiere aclaración. Una motivación clásica es la simplificación. Otra es la aproximación de una forma a otra. También podemos considerar la motivación de intentar algo sin motivo alguno más que probar suerte.

Notese que estas motivaciones tienen una prioridad implícita. Cualquier motivación que transforme la expresión origen presupone que las expresiones origen y destino son distintas. Es decir toda motivación salvo la primera, presupone que la primera no cumple.

Al mismo tiempo, se presupone que si se tiene la última motivación, probar algo sólo por probar suerte, es porque todas las otras motivaciones no tuvieron éxito. Es responsabilidad de la estrategia hacer explícito este orden de prioridad.

Cada una de estas motivaciones tiene relacionada una serie de reescrituras de las cuales hace uso y una manera de coordinarlas. Llamaremos tácticas a los conjuntos de reescrituras coordinadas por una única motivación.

4.1 Clasificación de las reescrituras por motivación

Agruparemos entonces las reescrituras en varios conjuntos. Primero consideraremos aquellas reescrituras que simplifican la expresión. Muchas de estas son obvias como $\alpha \Longrightarrow \neg\neg\alpha$, $\alpha \wedge \alpha \Longrightarrow \alpha$. Otras dependerán del concepto de simplicidad que describiremos más adelante. A este conjunto lo llamaremos "simplificaciones".

Segundo consideraremos las reescrituras que permiten transformar la conectiva principal de una expresión en otra. Por ejemplo, $\alpha \wedge (\beta \vee \gamma) \Longrightarrow ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$. Llamaremos a este conjunto "aproximaciones".

También podríamos considerar las reescrituras que transformar una expresión en otra más complicada. Sería el grupo complemento de las simplificaciones y lo llamaríamos "complificaciones". Intentaremos usar las reescrituras de este conjunto lo menos posible. A aquellas reescrituras de este conjunto que encontremos absolutamente necesario utilizar, las pondremos en un conjunto nuevo llamado "sacudidas".

Notar que el conjunto de aproximaciones puede compartir reescrituras tanto con las simplificaciones como con las complificaciones. Consideramos esto apropiado, ya que aunque se usa la operación, se usa con un propósito distinto y este propósito el que buscamos formalizar.

4.2 Esbozo de estrategia

Con lo que tenemos hasta ahora podemos presentar un esbozo de estrategia. Dado $\alpha \mathcal{Z} \beta$

Si es trivial que $\alpha = \beta$, se termina.

Si α puede ser simplificado en una expresión distinta α', se intenta demostrar $\alpha' \mathcal{Z} \beta$

Si β puede ser simplificado en una expresión distinta β', se intenta demostrar $\alpha \mathcal{Z} \beta'$

Si se puede aproximar α a β resultando en una fórmula α' distinta de α, se intenta demostrar con $\alpha' \mathcal{Z} \beta$

En cualquier otro caso, se intenta sacudir [1] α resultando en una fórmula α' distinta de α, y se intenta demostrar con $\alpha' \mathcal{Z} \beta$

Tenemos entonces cuatro tácticas. Trivializar, Simplificar, Aproximar y Sacudir. Vamos a intentar mantener las tácticas independientes entre sí. Sin embargo, la estrategia nos fuerza un orden de aplicación de las tácticas y por lo tanto tendremos cuidado a la hora de poner una reescritura y su inversa en tácticas distintas. Por ejemplo, dado que encontramos de gran interés tener la reescritura $\alpha \equiv \beta \Longrightarrow (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$ en la táctica sacudir, evitaremos tener la táctica $(\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta) \Longrightarrow \alpha \equiv \beta$ en simplificar. Dado que de otra manera, la estrategia entraría en un *loop* infinito.

4.3 Trivializar

La demostración más sencilla que vamos a admitir es aquella donde la fórmula origen y destino son trivialmente equivalentes.

Pero queda la pregunta de cuando las consideramos trivialmente equivalentes. La propuesta más sencilla sería considerar que solo las fórmulas que son sintácticamente iguales son trivialmente equivalentes. Pero esto se aleja del concepto que usado calculacionalmente.

Una segunda propuesta es, teniendo en cuenta de la asociatividad, conmutatividad de las conectivas binarias que usamos ($\wedge$, $\vee$, $\equiv$). Para ignorar

[1] aplicar una reescritura del conjunto de sacudidas

el orden de los elementos y los paréntesis. También podríamos ignorar los elementos repetidos en $\wedge$ y $\vee$. Pero entonces estaríamos entrando en el territorio de la simplificación.

Como punto intermedio, vamos a considerar que dos fórmulas son trivialmente equivalentes sin importar el orden de sus subexpresiones.

Además de equivalencias triviales, también podemos considerar las inequivalencias triviales, por ejemplo: $\alpha \mathcal{Z} \neg\alpha$. Es obvio que no podremos llegar del origen al destino y por lo tanto es un caso trivial.

También vamos a considerar trivial el caso $\alpha \mathcal{Z} \beta$ donde α y β son variables.

4.3.1 Formalización de la trivialidad

```
1 -- devuelve True si es una equivalencia  trivial
2 isTrivial::FL->FL->Bool
3 isTrivial (Var x) (Var y) = True
4 isTrivial x y = x==y || x==(negateOnce y)
5
6 negateOnce::FL->FL
7 negateOnce (N x) = x
8 negateOnce x = (N x)
```

4.3.2 Formalización de la igualdad

Definimos la igualdad de la siguiente manera. Dado a == b, primero pasamos a y b a una forma canónica. Haciendo uso de la función sort. Luego comparamos que las formas canónicas sean idénticas.

```
1 --Agregamos deriving (Ord), para poder hacer uso de Listing.sort
2 data FL = Top | Bot | Var Variable | N FL | C [FL] | D [FL] | E [FL
      ]
3       deriving (Ord)
4
5 instance Eq FL where{
6 (==) a b = structuralEquality (sort a) (sort b)};
7
8 structuralEquality::FL->FL->Bool
9 structuralEquality Top Top = True
```

```
10 structuralEquality Bot Bot = True
11 structuralEquality (Var x) (Var y) = x==y
12 structuralEquality (N x) (N y) = structuralEquality x y
13 structuralEquality (C x) (C y) = structuralEqualityLists x y
14 structuralEquality (D x) (D y) = structuralEqualityLists x y
15 structuralEquality (E x) (E y) = structuralEqualityLists x y
16 structuralEquality _ _ = False
17
18 structuralEqualityLists::[FL]->[FL]->Bool
19 structuralEqualityLists (x:xs) (y:ys) = (structuralEquality x y) &&
       (structuralEqualityLists xs ys)
20 structuralEqualityLists (x:xs) _ = False
21 structuralEqualityLists _ (y:ys) = False
22 structuralEqualityLists [] [] = True
23
24 sort::FL->FL
25 sort Top = Top
26 sort Bot = Bot
27 sort (Var x) = (Var x)
28 sort (N x) = (N (sort x))
29 sort (C y) = C (List.sort (map (sort) y))
30 sort (D y) = D (List.sort (map (sort) y))
31 sort (E y) = E (List.sort (map (sort) y))
```

4.3.3 Comienzo de la estrategia

Dado que tenemos isTrivial, podemos empezar a formalizar la estrategia en Haskell:

```
1  solve::FL->FL->Result (Proof FL)
2  solve p d
3      | trivial p d = trivialResult p d
4          | otherwise = Fail "No se logro demostrar"
5
6  trivialResult::FL->FL->Result (Proof FL)
7  trivialResult (Var x) (Var y)
8                 | x==y = (Success (Line (Var x)))
9                 | otherwise = Fail "variables distintas"
10 trivialResult x y
11         |x==y = (Success (Line x))
12         |x== (negateOnce y) = Fail "contradicci\'{o}n"
```

Dado que la estrategia no siempre tiene éxito, no puede devolver una demostración, sino que devuelve un resultado que puede o no ser una demostración. Permitiremos el resultado éxito que contiene una demostración y el resultado fracaso que contiene un string.

```
1 data Result a = Success a | Fail String
```

4.4 Simplificar

La simplificación es una táctica clásica que trata al origen y al destino por separado. Esta es una de sus grandes virtudes, al tratar al origen y al destino por separado es mucho más sencilla de comprender y de usar.

Otra gran virtud es que reduce el espacio de expresiones a considerar. Las siguientes tres fórmulas $\neg(p \wedge \neg p)$, $(p \vee \neg p)$ y $(p \equiv p)$, todas se simplifican en la misma fórmula $\top$. Esto significa que aún cuando tratamos al origen y al destino por separado, al simplificar ambas es muy factible que nos estemos aproximando.

Esta cualidad también sirve para preparar las fórmulas para las siguientes tácticas. Es decir, reduce el espacio de fórmulas que deben considerar el resto de las tácticas.

Por último, como indica su nombre, hace las expresiones más simples. Lo cual no es una virtud menor.

4.4.1 Criterio

Es fácil acordar que la fórmula p es más simple que su equivalente $p \vee p$. Una propuesta de simplificación, podría ser aplicar cualquier reescritura posible que resulte en una expresión de largo menor. Pero este no es el único criterio. Es tradición considerar que la fórmula $(\neg p \wedge \neg q)$ como una simplificación de $\neg(p \vee q)$. Por más de que tiene una extensión mayor. Esto se debe a que, en lo posible, preferimos que la negación este directamente sobre las variables, en vez de sobre un $\vee$ o un $\wedge$.

Queda la duda de que hacemos con las expresiones del estilo $\neg(p \equiv q)$. Podemos anidar el $\neg$ en estos casos, pero nos vemos forzados a elegir entre

$\neg p \equiv q$ y $p \equiv \neg q$. Para evitar tomar esta decisión preferiremos factorizar el $\neg$ siempre que aparezca dentro de una equivalencia.

El conjunto de simplificaciones tendría entonces todas las reescrituras que reducen la extensión de la fórmula con la excepciones:

Las reescrituras de de Morgan, donde se preferirán aquellas reescrituras que anidan el $\neg$. $(\neg(p \vee q) \Longrightarrow \neg p \wedge \neg q)$ y $(\neg(p \wedge q) \Longrightarrow \neg p \vee \neg q)$.

También agregaríamos aquellas reescrituras que factorizan el $\neg$ sobre la equivalencia. $(\neg p \equiv q \Longrightarrow \neg(p \equiv q))$ y $(p \equiv \neg q \Longrightarrow \neg(p \equiv q))$

Por ultimo, no tomaremos en cuenta la factorización del $\wedge$ y el $\vee$ $((p \wedge q) \vee (p \wedge r) \Longrightarrow p \wedge (q \vee r))$ ni las definiciones del $\equiv$ $((\neg p \vee q) \wedge (p \vee \neg q) \Longrightarrow p \equiv q$ y $(p \wedge q) \vee (\neg p \vee \neg q) \Longrightarrow p \equiv q)$. Esto se debe a que necesitaremos hacer uso del sentido inverso de dichas reescrituras en otras tácticas y si las admitiéramos cómo simplificaciones la estrategia entraría en *loop* infinito.

4.4.2 Estrategia con simplificación

```
1  solve::FL->FL->Result (Proof FL)
2  solve p d
3      | trivial p d = trivialResult p d
4      | p /= simp_p = addResults (Success simp_pproof) (solve simp_p
           d)
5      | d /= simp_d = addResults (solve p simp_d) (Success (
           simp_dproof))
6      | otherwise = Fail "No se logro demostrar"
7        where
8          simp_pproof = simplify p;
9          simp_dproof = (rev (simplify d));
10         simp_p = lastF simp_pproof;
11         simp_d = firstF simp_dproof;
```

La función simplificar recibe una fórmula y devuelve una demostración de que dicha fórmula es equivalente a una fórmula más simple. La estrategia intenta simplificar el origen, si lo logra busca demostrar que la fórmula origen simplificada es equivalente a la fórmula destino. Para luego concatenar las dos demostraciones.

Similarmente, si puede simplificar la fórmula destino, busca probar la equivalencia de el origen con la fórmula destino simplificada. Pero a la hora de unir las demostraciones, se requiere invertir el orden de la demostración de la simplificación del destino. Por eso, hacemos uso de la función rev, que invierte el orden de la demostración, inmediatamente que simplificamos el destino.

4.5 Aproximar

Por aproximar nos referimos a reescribir la fórmula origen en otra fórmula más próxima a la fórmula destino. La aproximación entonces depende del criterio de proximidad que decidamos. Aquí podríamos decidir, por ejemplo que las fórmulas sean del mismo largo. O que si cierta conectiva no aparece en la fórmula destino, tampoco debe aparecer en la fórmula resultante de aproximar el origen al destino.

Consideraremos que dos fórmulas son absolutamente próximas cuando son iguales. Concepto que ya definimos en detalle. Podemos usar la igualdad entonces como una base para diseñar la aproximación.

La igualdad compara primero las conectivas principales. Si coinciden llama recursivamente con los hijos, sino devuelve falso. Por analogía, la aproximación primero igualaría las conectivas principales y luego buscaría aproximar las subexpresiones.

El caso en el que ambas conectivas principales son la negación es sencillo. Dado que solo hay una subexpresión inmediata, dado $\neg\alpha\mathcal{Z}\neg\beta$, busca demostrar $\alpha\mathcal{Z}\beta$ y si encuentra una demostración construye una nueva agregando $\neg$ en cada paso. Esto es similar a lo que se hace en lápiz y papel, donde se copia la negación en cada paso pero se ignora en su totalidad.

El caso para las conectivas $\wedge$, $\vee$ y $\equiv$ es más complejo. Se pueden tener varias subexpresiones y nuestra definición de igualdad ignora el orden. Por ejemplo, en el caso aproximar D[α,β] a D[γ,δ], no basta con intentar aproximar α con γ y β con δ. Ya que éste es un orden circunstancial. Tampoco podemos ordenarlas de manera independiente, ya que son fórmulas distintas. La opción que nos queda es intentar igualar cada fórmula de [α,β] con una fórmula distinta de [γ,δ].

Dicho en otras palabras, si las conectivas principales de la fórmula origen y destino son iguales, se busca crear una relación uno a uno de equivalencia entre sus subfórmulas. Se revela la naturaleza recursiva de la aproximación.

Si se tiene éxito, se construirían varias demostraciones que luego deberían unirse.

Vamos a empezar por programar una función que busque probar una equivalencia entre una fórmula origen y alguna fórmula de una lista de destinos.

```
1 matchOne:: FL -> [FL] -> Result (Proof FL)
2 matchOne x (y:ys) = if isSuccess r then r else matchOne n x ys
3           where r =solve x y
4 matchOne x [] = Fail "No se encontró una fórmula equivalente"
```

Ahora podemos programar una función que dada dos listas de fórmulas intente igualar cada fórmula de la primera lista con una distinta de la segunda lista.

```
1 matchAll::[FL]->[FL]->[Result (Proof FL)]
2 matchAll (x:xs) (y:ys) = case r of {Success p -> r:(matchAll xs (
     List.delete (lastF p) (y:ys))); _->[Fail "no se encontró una fó
     rmula equivalente"];}
3           where
4             r = (matchOne x (y:ys))
5
6 matchAll (x:xs) [] = [Fail "sobran orígenes"]
7 matchAll [] (y:ys) = [Fail "sobran destinos"]
8 matchAll [] [] = []
```

Falta decidir como unir la lista de demostraciones. Volviendo al caso de D[α,β] a D[γ,δ]. Si encontramos una demostración de que $\alpha = \delta$ y $\beta = \gamma$, nos gustaría crear una nueva demostración donde la conectiva principal es siempre D se realizan todos los pasos para transformar α en δ copiando β intacto y luego se realizan todos los pasos para transformar β en γ copiando δ intacto. A la función que crea demostraciones de ese estilo le llamamos sumParallelResults.

Ahora podemos programar que como aproximar si las conectivas son iguales:

```
1 matchSons::Int->FL->FL->Result (Proof FL)
2 matchSons n (C s) (C r) = sumParallelResults (matchAll n (s) (r)) (
    C)
3 matchSons n (D s) (D r) = sumParallelResults (matchAll n ( s) ( r))
     (D)
4 matchSons n (E s) (E r) = sumParallelResults (matchAll n ( s) ( r))
     (E)
5 matchSons n (N x) (N y) = mapIfSuccess (N) (solve n x y)
```

Por último falta decidir que hacer si las conectivas son distintas. Dado que las dos expresiones ya están simplificadas, no vamos a preocuparnos de aproximar la negación a otra conectiva. Tampoco vamos a preocuparnos de aproximar $\vee$ a $\equiv$ o $\wedge$ a $\equiv$, dependiendo en estos casos en sacudir. Solo aproximaremos los siguientes casos:

```
1 --devuelve True si se puede aproximar
2 approximable (C x) (D y) = distributableCoD (C x)
3 approximable (D x) (C y) = distributableDoC (D x)
4 approximable (E x) (C y) = True
5 approximable (E x) (D y) = True
6 approximable _ _ = False
7
8 --aproxima
9 approximate (C x) (D y) = distributeCoD (C x)
10 approximate (D x) (C y) = distributeDoC (D x)
11 approximate (E x) (C y) = defEinC (E x)
12 approximate (E x) (D y) = defEinD (E x)
13
14 --devuelve la  justificaci ón de la aproximación
15 approxMotive (N x) (y) = "Aproximar: anidar negación"
16 approxMotive (C x) (D y) = "Aproximar: conjunción a disyunción"
17 approxMotive (D x) (C y) = "Aproximar: disyunción a conjunción"
18 approxMotive (E x) (C y) = "Aproximar: equivalencia a conjunción"
19 approxMotive (E x) (D y) = "Aproximar: equivalencia a disyunción"
```

4.5.1 Estrategia con aproximación

```
1 solve::FL->FL->Result (Proof FL)
2 solve p d
3   | isTrivial p d = trivialResult p d
4   | p /= simp_p = addResults (Success simp_pproof) (solve simp_p d)
5   | d /= simp_d = addResults (solve p simp_d) (Success (simp_dproof
       ))
6   | not (eqr) && approximable p d && isSuccess approx = ifSuccess (
       addStep p approxmotive) approx
7   | eqr && (isSuccess sons) = sons
8   | otherwise = Fail "No se logro demostrar"
9         where
10          simp_pproof = simplify p;
11          simp_dproof = (rev (simplify d));
12          simp_p = lastF simp_pproof;
13          simp_d = firstF simp_dproof;
14          approx = solve (approximate p d) d;
15          approxmotive = approxMotive p d;
16          eqr = (equalRoot p d);
17          sons = matchSons n p d
```

4.6 Sacudir

Por sacudir, nos referimos a los pasos que realizamos cuando no se puede avanzar con otra táctica. Las sacudidas no simplifica, ni aproxima fórmulas, su única virtud consiste en cambiar la fórmula por una nueva que todavía no tratamos. Es necesario sacudir cuando no se puede avanzar con ninguna táctica anterior, con la esperanza de que una vez sacudida la fórmula se pueda construir una demostración.

Debemos observar que no nos sirve cualquier sacudida, si sacudimos α en α', α no puede ser una simplificación de α'. De lo contrario la estrategia quedaría en *loop* infinito. Debemos tener la mismas consideración con la aproximación.

Teniendo en cuenta esta consideración, permitiremos las siguientes sacudidas: la distribución del $\wedge$ sobre el $\vee$ ($p \wedge (q \vee r) \Longrightarrow ((p \wedge q) \vee (p \wedge r))$), la distribución del $\vee$ sobre el $\wedge$ ($p \vee (q \wedge r) \Longrightarrow ((p \vee q) \wedge (p \vee r))$) y las definiciones del $\equiv$ ($((\neg p \vee q) \wedge (p \vee \neg q) \Longrightarrow p \equiv q$ y $(p \wedge q) \vee (\neg p \vee \neg q) \Longrightarrow p \equiv q$)

Formalizado en Haskell:

```
1 --devuelve True si se puede sacudir
2 shakeable (C a) = distributableCoD (C a)
3 shakeable (D a) = distributableDoC (D a)
4 shakeable (E a) = True
5 shakeable (N(E a)) = True
6 shakeable _ = False
7
8 --sacude
9 shake (C a) = distributeCoD (C a)
10 shake (D a) = distributeDoC (D a)
11 shake (E a) = defEinC (E a)
12 shake (N(E a)) = N(defEinC (E a))
13
14 --devuelve la justificaci ón de la sacudida
15 shakeMotive (C a) = "Sacudir: conjunción"
16 shakeMotive (D a) = "Sacudir: disyunción"
17 shakeMotive (E a) = "Sacudir: equivalencia"
```

Es interesante notar que, en principio, no hay limite a la cantidad de veces que se pueda sacudir una fórmula. Esto impediría que la estrategia termine cuando se busca demostrar fórmulas que no son equivalentes. Una solución, es agregar un parámetro a la estrategia que indica cuantas veces se puede aplicar la táctica sacudir en una misma demostración. A este parámetro lo llamamos, apropiadamente, "paciencia".

La táctica sacudir sería: si la paciencia es mayor a cero y el origen puede ser sacudido, sacudo el origen y busco demostrar recursivamente con el nuevo origen y una paciencia menor. si la paciencia es mayor a cero y el destino puede ser sacudido, sacudo el destino y busco demostrar recursivamente con el nuevo destino y una paciencia menor.

4.6.1 Estrategia con sacudir

```
1 solve::Int->FL->FL->Result (Proof FL)
2 solve n p d
```

```
3     | isTrivial p d = trivialResult p d
4     | p /= simp_p = addResults (Success simp_pproof) (solve n
        simp_p d)
5     | d /= simp_d = addResults (solve n p simp_d) (Success (
        simp_dproof))
6     | not (eqr) && approximable p d && isSuccess approx = ifSuccess
          (addStep p approxmotive) approx
7     | eqr && (isSuccess sons) = sons
8     | n>0 && (shakeable p) && (isSuccess shakep) = ifSuccess (
        addStep p shakepmotive) shakep
9     | n>0 && (shakeable d) && (isSuccess shaked) =  ifSuccess (
        addFinalStep d shakedmotive) shaked
10    | otherwise = Fail "No se logro demostrar"
11        where
12          simp_pproof = simplify p;
13          simp_dproof = (rev (simplify d));
14          simp_p = lastF simp_pproof;
15          simp_d = firstF simp_dproof;
16          approx = solve n (approximate p d) d;
17          approxmotive = approxMotive p d;
18          eqr = (equalRoot p d);
19          sons = matchSons n p d
20          shakep = solve (n-1) (shake p) d
21          shaked = solve (n-1) p (shake d)
22          shakepmotive = shakeMotive p
23          shakedmotive = shakeMotive d
```

5 Puesta a prueba

Una vez formalizada la estrategia, nos interesa ponerla a prueba. Es interesante ver que demostraciones logra construir y cómo. Las equivalencias que no logra demostrar, también son interesantes, ya que señalan las posibles mejoras que se le podrían realizar a la estrategia.

Dado que la estrategia a veces realiza pasos repetitivos, crearemos una función nueva que primero demuestra y luego quita los pasos repetitivos. La misma hace uso de la función removeRedundantSteps que ya definimos.

```
1 solveShort::Int->FL->FL->Result (Proof FL)
2 solveShort n p d = ifSuccess (removeRedundantSteps) (solve n p d)
```

5.1 Regla de oro

La tautología $\alpha \equiv \beta \equiv (\alpha \vee \beta) \equiv (\alpha \wedge \beta)$ es una que suele sorprender y que resulta muy útil en las demostraciones calculacionales. Tanto es así, que Dijkstra la denomina la regla de oro.[3][9]

Dijkstra no acostumbra a demostrarla, sino que la estipula. Pero supongamos que es la primera vez que vemos esta tautología y que queremos demostrarla. Dado que $\equiv$ es asociativo, podríamos intentar demostrar $\alpha \equiv \beta \mathcal{Z} (\alpha \vee \beta) \equiv (\alpha \wedge \beta)$ o $\alpha \equiv (\alpha \vee \beta) \mathcal{Z} \beta \equiv (\alpha \wedge \beta)$ o $\alpha \mathcal{Z} \beta \equiv (\alpha \vee \beta) \equiv (\alpha \wedge \beta)$ por mencionar algunos casos.

Probemos con $\alpha \equiv \beta \mathcal{Z} (\alpha \vee \beta) \equiv (\alpha \wedge \beta)$

```
1 solveShort 4 (E[Var Xa, Var Xb]) (E[D[Var Xa, Var Xb], C[Var Xa,
     Var Xb]])
```

Zorro devuelve la siguiente demostración:

$a \equiv b$
$= \{\downarrow$ Sacudir: equivalencia$\}$
$(\neg a \vee b) \wedge (a \vee \neg b)$
$= \{\downarrow$ Aproximar: conjunción a disyunción$\}$
$(\neg a \wedge (a \vee \neg b)) \vee (b \wedge (a \vee \neg b))$
$= \{\downarrow$ Sacudir: conjunción$\}$
$((a \wedge \neg a) \vee (\neg b \wedge \neg a)) \vee (b \wedge (a \vee \neg b))$
$= \{\downarrow$ Opuestos$\}$
$(\bot \vee (\neg b \wedge \neg a)) \vee (b \wedge (a \vee \neg b))$
$= \{\downarrow$ Neutro$\}$
$(\neg b \wedge \neg a) \vee (b \wedge (a \vee \neg b))$
$= \{\downarrow$ Sacudir: conjunción$\}$
$(\neg b \wedge \neg a) \vee ((a \wedge b) \vee (\neg b \wedge b))$
$= \{\downarrow$ Opuestos$\}$
$(\neg b \wedge \neg a) \vee ((a \wedge b) \vee \bot)$
$= \{\downarrow$ Neutro$\}$
$(a \wedge b) \vee (\neg a \wedge \neg b)$
$= \{\uparrow$ Neutro$\}$
$\top \wedge ((a \wedge b) \vee (\neg a \wedge \neg b))$
$= \{\uparrow$ de Morgan$\}$
$\top \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ Opuestos$\}$
$(\neg a \vee \neg b \vee a \vee b) \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ Asociativa$\}$
$((\neg a \vee \neg b) \vee (a \vee b)) \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ de Morgan$\}$
$(\neg(a \wedge b) \vee (a \vee b)) \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ Sacudir: equivalencia$\}$
$(a \wedge b) \equiv (a \vee b)$

Es interesante notar que si probamos con una paciencia menor, la demostración resulta más corta:

```
1 solveShort 2 (E[Var Xa, Var Xb]) (E[D[Var Xa, Var Xb], C[Var Xa,
    Var Xb]])
```

Zorro devuelve la siguiente demostración:

$a \equiv b$
$= \{\downarrow$ Aproximar: equivalencia a disyunción$\}$
$(a \wedge b) \vee (\neg a \wedge \neg b)$
$= \{\uparrow$ Neutro$\}$
$\top \wedge ((a \wedge b) \vee (\neg a \wedge \neg b))$
$= \{\uparrow$ de Morgan$\}$
$\top \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ Opuestos$\}$
$(\neg a \vee \neg b \vee a \vee b) \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ Asociativa$\}$
$((\neg a \vee \neg b) \vee (a \vee b)) \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ de Morgan$\}$
$(\neg(a \wedge b) \vee (a \vee b)) \wedge ((a \wedge b) \vee \neg(a \vee b))$
$= \{\uparrow$ Sacudir: equivalencia$\}$
$(a \wedge b) \equiv (a \vee b)$

Al menos en este caso, la impaciencia parece tener sus beneficios.

Leamos con detenimiento la segunda demostración, que es más corta. Primero observamos que no basto con simplificar $a \equiv b$, ya que el primer paso es una aproximación. Pero si observamos, es una aproximación a una fórmula distinta a la fórmula destino. Esto significa que primero se intento sacudir al fórmula origen, se fracaso (por falta de paciencia) y se intento sacudir la destino.

Leamos que sucede con la fórmula destino. Primero se sacude $(\neg(a \wedge b) \vee (a \vee b)) \wedge ((a \wedge b) \vee \neg(a \vee b))$y luego se simplifica hasta llegar a $(a \wedge b) \vee (\neg a \wedge \neg b)$. Al llegar a esta fórmula solo falta probar que $a \equiv b$ es equivalente a $(a \wedge b) \vee (\neg a \wedge \neg b)$, lo que sólo requiere una aproximación.

5.2 Comparación con demostraciones de Dijkstra

Las siguientes demostraciones son hechas en comparación con las demostraciones de Dijkstra en [3].

5.2.1 Primera demostración

La demostración de que $x \equiv \neg x$ es equivalente a $\bot$ queda igual a la demostración de Dijkstra.

```
1 solve 2 (E[Var Xx, N(Var Xx)]) (Bot)
```

$x \equiv \neg x$
$= \{\downarrow \text{ parity of } =\}$
$\neg(x \equiv x)$
$= \{\downarrow \text{ Multiplicidad}\}$
$\neg\top$
$= \{\downarrow \text{ Def Bot}\}$
$\bot$

5.2.2 Segunda demostración

La demostración de que $x \equiv \neg x$ es equivalente a $\bot$ también queda igual a la demostración de Dijkstra.

```
1 solveShort 2 (C[E[Var Xx, Var Xy], E[N(Var Xx), N(Var Xy)]]) (E[Var
      Xx, Var Xy])
```

$(x \equiv y) \wedge (\neg x \equiv \neg y)$
$= \{\downarrow \text{ parity of } =\}$
$(x \equiv y) \wedge (x \equiv y)$
$= \{\downarrow \text{ Multiplicidad}\}$
$x \equiv y$

5.2.3 Tercera demostración

La demostración de que $(x \wedge \bot) \equiv \bot$ es equivalente a $\top$ queda muy distinta a la de Dijkstra.

```
1 solveShort 5 (E[C[Var Xx, Bot], Bot]) Top
```

$(x \wedge \bot) \equiv \bot$
$= \{\downarrow \text{ Absorción}\}$

$\bot \equiv \bot$
$=$ {$\downarrow$ Multiplicidad}
$\top$

Mientras que la demostración de Dijkstra es:

$(x \wedge \bot) \equiv \bot$
$=$ {Regla dorada}
$(x \vee \bot) \equiv x$
$=$ {Neutro de la disyunción}
$\top$

La distinción entre las demostraciones, primero permite hacer uso de la regla dorada directamente y segundo permite sustituir el axioma $x \vee \bot) \equiv x$ por el valor $\top$ directamente, en vez de simplificarlo por pasos.

5.3 Dificultad obvia

Supongamos que queremos demostrar que $a \equiv ((\neg b \vee c) \wedge (b \vee \neg c))$ es equivalente a $c \equiv ((\neg a \vee b) \wedge (a \vee \neg b))$. Para nosotros esto resulta sencillo de demostrar, ya que basta con aplicar la definición de $\equiv$ en ambas fórmulas para conseguir $a \equiv b \equiv c$ y $c \equiv a \equiv b$. Que son trivialmente equivalente.

Sin embargo, la estrategia no puede aplicar la reescritura $(\neg a \vee b) \wedge (a \vee \neg b) \Longrightarrow a \equiv b$ dado que usamos la reescritura inversa al sacudir y por lo tanto ese camino queda descartado.

La estrategia en cambio intenta sacudir el $\equiv$ en ambos, resultando en dos fórmulas mucho más largas. Esto claramente no es elegante. Una mejora futura, podría ser que el sistema tenga cierta memoria de que reescrituras hizo, para así poder permitir $(\neg a \vee b) \wedge (a \vee \neg b) \Longrightarrow a \equiv b$, como simplificación sin riesgo de entrar en *loop* infinito.

```
1 solve 3 (E[ Var Xa, C[D[N(Var Xb), (Var Xc)], D[(Var Xb), N(Var Xc)
    ]]]) (E[ Var Xc, C[D[N(Var Xa), (Var Xb)], D[(Var Xa), N(Var Xb)
    ]]])
```

```
1 Fail: No se logro demostrar
```

6 Trabajo futuro

Como ya hemos mencionado, vemos este trabajo como una primera incursión en la formalización de la metodología matemática. A futuro nos gustaría extenderlo para que incluya a la lógica de predicados.

Sin embargo, hay otras extensiones más inmediatas que también nos interesan.

Nos gustaría formalizar un método para pasar de una fórmula a otra que cumpla una forma. Es decir, nos gustaría tener un método que dado una fórmula origen y un conjunto de fórmulas destino, intente demostrar que la fórmula origen es equivalente a alguna de las fórmulas destino.

La versión más simple de esta extensión, es una variación de la función matchOne que vimos previamente. Pero matchOne intenta demostrar cada equivalencia individualmente, cuando esto no es siempre necesario. Si las fórmulas destino compartieran la misma fórmula normal, nos gustaría que el método aproxime la fórmula origen a esa conectiva antes de probar cada fórmula destino por separado.

De lograr esto, se podría sustituir matchOne en la aproximación por este nueva formalización y se conseguirían demostraciones mucho más elegantes.

7 Palabras finales

Al comenzar la investigación teníamos grandes ansías y ambiciones. Al hablar con el tutor de la tesis, decidimos, a regañadientes, plantearnos un objetivo mucho más concreto.

Durante la realización de esta tesis descubrimos que ese objetivo concreto era mucho más rico de lo que sospechábamos y que planteaba grandes desafíos.

Seguimos persiguiendo la ambición original, pero perdimos la ansiedad. El camino hasta el objetivo es muy largo, pero, evidentemente, es un camino muy disfrutable.

8 Referencias Bibliográficas

[1] E. W. Dijkstra, "0 preface (mathematical methodology)," *EWD Manuscripts*, 1989.

[2] ——, "Introducing a course on the design and use of calculi," *EWD Manuscripts*, 1992.

[3] *Predicate Calculus And Program Semantics.* Springer-Verlag, 1989.

[4] E. W. Dijkstra, "On well-shaped mathematical arguments," *EWD Manuscripts*, 1980.

[5] ——, "A book review," *EWD Manuscripts*, 1982.

[6] ——, "For brevity's sake," *EWD Manuscripts*, 1989.

[7] *Predicate Calculus And Program Semantics.* Springer-Verlag, 1993.

[8] E. W. Dijkstra, "On anthropomorphism in science," *EWD Manuscripts*, 1985.

[9] ——, "The everywhere operator once more," *EWD Manuscripts*, 1990.

9 Anexos

9.1 Manual del programa

Instrucciones para ejecutar el programa:

- Instalar Glasgow Haskell Compiler (que puede obtener se en http://www.haskell.org/ghc/)
- Ejecutar Glasgow Haskell Compiler
- Dentro del compilador introducir el comando ":cd /Zorro" donde /Zorro es la carpeta donde se encuentra el código fuente.
- Ejecutar el comando ":load Zorro.hs"
- Ya se puede ejecutar el programa. A modo ejemplo, se puede probar con: solveShort 2 (C[E[Var Xx, Var Xy], E[N(Var Xx), N(Var Xy)]]) (E[Var Xx, Var Xy])

9.2 Código fuente

El código fuente se encuentra adjunta en la carpeta Anexos.