UNIVERSIDAD NACIONAL AUTONOMA DE MÉXICO

Posgrado en Ciencias e Ingeniería de la Computación

"SURREALIST: Un Método de Razonamiento Científico"

T E S I S

Que para obtener el grado de

Maestro en Ciencias
(Computación)

P r e s e n t a :

Elías Samra Hassán

D i r e c t o r a d e t e s i s :

Dra. Atocha Aliseda Llera

México, D.F. 2006

A mi mamá: Sofía Hassán Levy

Agradecimientos

Al entrar al posgrado mi expectativa era únicamente aprender lógica, semántica formal de programas, especificación formal de programas y representación del conocimiento.

Gracias a mi tutora la Dra. Atocha Aliseda Llera y a su invitación al seminario que organiza en el Instituto de Investigaciones Filosóficas sobre temas relacionados con la epistemología, ciencia cognitiva y filosofía computacional de la ciencia, entendí que el papel de la lógica no solo es deductivo sino que hay otro tipo de inferencias.

Fue muy enriquecedor escuchar a los Dres. Salma Saab, Axel Barceló, Raymundo Morado, Francisco Hernández, Silvio Pinto y por supuesto a la Dra. Aliseda, además de los invitados de la talla de Johan van Benthem y Robert Kowalski. Lo anterior aunado a las clases de los Dres. David Rosenblueth, Francisco Hernández, Vladislav Khartchenko, Carlos Velarde, Héctor Benítez, Raymundo Morado, Zbigniew Oziewicz, Amparo López y de nuevo sin olvidar a la Dra. Atocha Aliseda, cambiaron mi visión de lo que es la computación y su relación con la filosofía.

No es posible estudiar y mucho menos concluir una tesis sin los servicios de una biblioteca tan completa como la del IIMAS que además es atendida por un personal muy amable, a quienes agradezco enormemente las atenciones que tuvieron conmigo, muy especialmente a Juanita Alcalá, Mary Del Prado, Lety López Huerta y Cecilia Uribe.

También en forma muy especial quiero agradecer a mis amigos César Antonio Aguilar y Olga Acosta por su enorme ayuda en la corrección de estilo de mi tesis, lo que me ayudo a mejorar redacción.

Tuve varios tropiezos con mi computadora pero siempre tuve apoyo de mis compañeros, amigos y colegas por lo que quiero agradecerles a Iván Mejía por prestarme su monitor, a Alfredo Cobián de la Facultad de Ciencias por ayudarme a rescatar mi disco dañado por cortesía de la Compañía de Luz y Fuerza del Centro, así como a Dante Ortiz quien muchas veces me saco del apuro y me dio varios tips con Linux cuando hacía mis pininos y a Ana Cecilia Perez que también es experta linuxera.

Pidiendo perdón por las omisiones involuntarias quiero también agradecer a mis amigos y compañeros tanto del posgrado en computación como el de filosofía de la ciencia con quienes me tocó estudiar o tener discusiones muy enriquecedoras: Areli Rosas, Sun Guo Hua, Tania Pérez, Patricia Ibarra, Raymundo Santillan, Martín Solís, Alejandro Mancilla, Armando Cuellar, Antonio Neme, Ulianov, Mauricio, Fernando Robles, Erika y Alexei.

Otro privilegio del posgrado es la amabilidad del personal, gracias a Lulú, Violeta, Diana, Amalia por su ayuda con todos los trámites y a Juanita por

su amabilidad y eficiencia y calidad en el fotocopiado y a Álvaro ahora en lugar de Dante como soporte técnico.

Índice general

Resumen

En esta tesis escribí SURREALIST, un método heurístico que descubre leyes a partir de datos. Se escribió en el contexto de la *filosofía computacional de al ciencia* — un enfoque de la *filosofía de la ciencia* que sostiene la creencia de que los descubrimientos científicos cotidianos provienen de la aplicación de métodos generales de resolución de problemas.

GLAUBER, el predecesor de SURREALIST, es uno de los métodos escritos por Pat Langley, Herbert Simon y colaboradores para mostrar cómo algunos descubrimientos históricos pueden lograrse mediante la aplicación de programas de computadora basados en métodos generales de resolución de problemas a los datos disponibles en la época del descubrimiento. Este redescubre la ley de las reacciones químicas entre ácidos y álcalis produciendo sales.

SURREALIST se basa en el *álgebra relacional*. Con esa representación se puede operar todo el conjunto de cualidades a la vez lo que no era posible en GLAUBER. De esta manera SURREALIST forma clases de equivalencia de substancias más precisas que producen *todos* los hechos dados y mejores predicciones que GLAUBER. Hay una relación entre las formas normales en la *teoría de bases de datos* y la cuantificación en el *cálculo de predicados de primer orden* y SURREALIST toma ventaja de ésta para inferir la cuantificación correcta de las leyes descubiertas.

Para poder discutir y experimentar con GLAUBER, lo implementé en `Haskell` y ejecuté el ejemplo dado por sus autores para mostrar cómo descubre la ley de las reacciones químicas entre ácidos y álcalis. También incluí un apéndice que muestra como aplicar SURREALIST mediante consultas en `SQL` a un ejemplo extendido de reacciones químicas que GLAUBER no puede manejar y que SURREALIST si. Además de que SURREALIST descubre las leyes y las clases de equivalencia correctas.

Abstract

This thesis is about SURREALIST, an heuristic method that discovers laws from data. It was written in the context of *computational philosophy of science* — an approach of philosophy of science supporting the belief that everyday scientific discovery comes from the application of general *problem solving* methods.

GLAUBER, SURREALIST's predecessor, is one of various methods written by Pat Langley, Herbert Simon, *et al*, to show how some historical discoveries can be achieved via computer programs based in general problem solving methods applied to the data available at the time of the actual discovery. It rediscovers the law of chemical reactions between acids and alkalis producing salts.

SURREALIST us based on *relational algebra*. With this representation it is possible to operate the whole quality set at a time which was not possible in GLAUBER. In this way SURREALIST forms more accurate equivalence classes of substances producing *all* the given facts and better predictions than GLAUBER. There is a relationship between normal forms in *database theory* and quantification in *first order predicate calculus* and SURREALIST takes advantage of this to infer the correct quantification of the discovered laws.

In order to discuss and experiment width GLAUBER, I implemented it in `Haskell` and executed the example given by its authors for showing how it discovers the law of chemical reactions between alkalis and acids. I also included an appendix showing how to apply SURREALIST via `SQL` queries to an extended example of chemical reactions that GLAUBER can not handle but SURREALIST can. Additionally, SURREALIST discovers the correct laws and equivalence classes.

Introducción

En esta tesis se presenta SURREALIST un método de razonamiento científico. La idea de que todos los científicos siguen "El Método Científico" es muy popular entre los legos. Sin embargo, tanto los filósofos de la ciencia como los científicos sabemos que no existe un único método en la ciencia. Ni siquiera existe consenso en que se requiera de un método científico dado que hay algunas corrientes de pensamiento que creen, o al menos no descartan, la espontaneidad en el origen de algunas teorías revolucionarias.

El método que se presenta en esta tesis se basa en la creencia de que la ciencia cotidiana, y no necesariamente la revolucionaria, puede echar mano de los métodos que aporta la disciplina de *Resolución de Problemas*. Esa idea es de Herbert A. Simon y sus colaboradores quienes en Langley, Simon, Bradshaw, y Zytkow [1987] presentan varios programas que redescubren teorías científicas mediante métodos de inferencia inductiva aplicándolos a los datos conocidos en el momento histórico del descubrimiento.

Uno de los programas que presentan es GLAUBER, nombrado así en honor de alquimista del siglo XVII Johann Rudolf Glauber (1604–1670)[1]. Su tarea es descubrir la ley de las reacciones mediante las cuales se forma una sal a partir de un ácido y un álcali. Los datos que procesa son dos predicados. El primero `Has-Quality`, corresponde a oraciones de la forma: "La substancia vitriolo tiene la cualidad sabor con el valor agrio". El segundo predicado, `Reacts`, corresponde a oraciones de la forma: "La reacción en la que se vierten las substancias vitriolo y sosa cáustica, tiene como resultado la(s) substancia(s) sal de Glauber".

GLAUBER aplica el método de resolución de problemas implementado en PRISM, el sistema en que fue escrito, para clasificar las substancias mediante las cualidades expresadas en el predicado `Has-Quality` y generaliza el predicado `Reacts` substituyendo a las substancias en los predicados por variables cuantificadas sobre las clases descubiertas.

[1] Realizó importantes descubrimientos en lo relativo a las reacciones iónicas. Algunas aportaciones comprenden el haber sido el primero en producir ácido clorhídrico (HCl). El sulfato de sodio (Na_2SO_4), una de las substancias que descubrió, se conoce también como la "*sal de Glauber*".

Langley, Simon y colaboradores evalúan la calidad de la teoría descubierta mediante lo que denominan su *poder predictivo*, es decir, por su capacidad para generar los datos de los que se indujo y por su *potencial predictivo*, o sea la capacidad de predecir lo que todavía no se ha observado.

Los experimentos presentados en Langley *et al.* [1987] tienen datos muy a la medida, aún así GLAUBER tiene problemas para cuantificar adecuadamente las leyes que infiere. Al complicar el experimento agregando reacciones entre *ácidos fuertes* y *débiles*, GLAUBER no puede encontrar las leyes adecuadas.

En esta tesis se escribió el método SURREALIST, que a partir de los mismos predicados (`Has-Quality` y `Reacts`) encuentra una teoría con mayor poder predictivo gracias a que representa la información mediante relaciones. El método de búsqueda consiste en dividir las substancias en clases adecuadas para describir las reacciones hasta describir todos los datos de donde se inducen. Si no tiene un *poder predictivo* del 100 % es una señal que indica al investigador que falta información acerca de las cualidades de las substancias, información que el investigador pudo considerar como no relevante o que todavía no es capaz de observar.

(muchas veces describir)

Una vez que se agregan las cualidades faltantes, que son relevantes para obtener una teoría con un *poder predictivo* del 100 %, se espera que su *potencial predictivo* también sea del 100 %, pero esto solo puede saberse hasta realizar nuevos experimentos, si la teoría fracasa en predecir nuevos resultados es el momento de revisarla agregando nuevas cualidades para que la teoría tenga un *poder predictivo* del 100 % ahora con los nuevos datos experimentales y las nuevas cualidades. Siendo ese el ciclo de vida de la experimentación y revisión de la teoría.

(corrección: no se puede conocer a priori)

Descripción del contenido

El primer capítulo ofrece una breve descripción del enfoque computacional de la filosofía de la ciencia, el descubrimiento científico como resolución de problemas, los métodos heurísticos en su graduación de débiles y fuertes, los algoritmos son métodos heurísticos fuertes además de las distintas estrategias para diseñarlos, terminando con una sección sobre inferencia inductiva donde se describe brevemente el programa BACON.1.

En el segundo capítulo se presenta GLAUBER, se describe el procedimiento, se discuten sus limitaciones se describe una implementación que mejora la determinación de cuantificadores y por último se muestra un ejemplo en el que GLAUBER fracasa. Este capítulo se complementa con el apéndice A en donde se presenta una implementación de GLAUBER escrita en `Haskell` con un estilo literario. La primera parte es una breve descripción de `Haskell`, se-

guida de la definición de los tipos de datos, por las funciones que implementan los procedimientos y termina con la ejecución de un ejemplo.

El tercer capítulo describe a SURREALIST, el principal producto de esta tesis. Las primeras secciones se dedican a las ideas detrás del método para generalizar. En la subsección §3.2.1 se describe la manera en que se transforman los predicados a su representación mediante relaciones. En §3.2.2 se describe el método para la búsqueda de clases. Es posible obtener más información con SURREALIST, en §3.2.3 se esbozan algunas ideas para un trabajo futuro. La sección 3.3 presenta el método para cuantificar las reglas.El método se basa en la relación entre las dependencias funcionales de una base de datos relacional y la cuantificación de predicados, curiosamente esta relación no aparece en textos de bases de datos. El capítulo termina con un resumen del método SURREALIST. Los apéndices B y C complementan al capítulo. El apéndice B presenta una síntesis de los conceptos básicos del álgebra relacional. En el apéndice C presenta una sesión en el intérprete `psql` de `PostgreSQL` aplicando interactivamente el método SURREALIST.

En el epílogo se presenta un resumen y algunas posibles lineas para continuar el trabajo.

Capítulo 1

Filosofía computacional de la ciencia

1.1 Filosofía computacional de la ciencia

La *filosofía de la ciencia con un enfoque computacional*, o *filosofía computacional de la ciencia*, pretende simular descubrimientos históricos por medio de programas, basándose, naturalmente, en los conocimientos disponibles en la época del descubrimiento.

Dentro de un enfoque computacional no se ve a la actividad científica como algo fortuíto lleno de inspiración creadora que de la nada plantea cambios revolucionarios. Por el contrario, se considera a la ciencia como una actividad metódica donde el científico busca una explicación a los fenómenos observados.

Langley, Simon, Bradshaw, y Zytkow [1987] plantean las siguientes ventajas del enfoque computacional de la filosofía de la ciencia:

1. *Se investiga la psicología del proceso del descubrimiento y los mecanismos probados para proceso de información.*
2. *Brinda algunas bases de la teoría normativa del descubrimiento. Propone y evalúa un número substancial de heurísticas diseñadas para facilitar el descubrimiento.*
3. *Reexamina las relaciones entre el proceso del descubrimiento y el proceso de verificación.*
4. *Dado que los ejemplos se toman de la historia de la ciencia, sugiere una metodología para examinar la historia de los descubrimientos, como una opción más para los historiadores de la ciencia.*

Uno de los pioneros de esta área es Simon, quien también es reconocido por sus aportaciones a la *resolución de problemas* e *inteligencia artificial* (véase Newell y Simon [1972, 1963], Simon [1978].) Junto con sus colaboradores (véase Langley *et al.* [1987]) sostiene el siguiente lema:

Descubrimiento científico = resolución de problemas

Esto es: el descubrimiento puede hacerse aplicando los métodos de resolución de problemas a la información y teoría existente, obteniéndose por respuesta una ley científica. Dicho enfoque es factible cuando el problema esta bien estructurado, lo que de alguna manera nos hace sentir más seguros de que entendemos la causalidad del problema, a diferencia de métocos como los de "caja negra" como las redes neuronales donde se infiere una función mediante el ajuste de una matriz de pesos (véase Frank, Kraiss, y Kuhlen [1998], Maass [1993]). En palabras de Simon, las condiciones que definen en qué grado un problema está bien estructurado depende de cuáles de las siguientes condiciones se cumplan:

- *Existe un criterio definido para probar cualquier solución, y hay un proceso mecanizable para la aplicación del criterio.*
- *Existe al menos un espacio del problema en el que puede representarse el estado inicial del problema, el estado objetivo, y todos otros estados que pueden alcanzarse o considerarse en el curso del intento de una solución al problema.*
- *Los cambios de estado alcanzables (los movimientos legales) pueden representarse en el espacio del problema como transiciones desde los estados dados hasta los estados directamente alcanzables desde ellos. Los movimientos considerables, ya sean legales o no, también pueden representarse.*
- *Cualquier conocimiento acerca del problema en cuestión, que adquiera la persona o el programa que busque su solución, es posible representarlo en uno o más espacios de problema.*
- *Todas esas condiciones mantienen el sentido fuerte de que los procesos básicos postulados solo requieran una cantidad de cómputo que pueda ser concretada, y la información postulada esté efectivamente disponible para los procesos (esto es, disponible solamente con la ayuda de cantidades de búsqueda que puedan ser llevadas a la práctica).*

Simon [1973] también citado en Langley et al. [1987] páginas 14-15.

Cuando los problemas no son bien estructurados, es posible abordarlos con otros enfoques, por ejemplo usando redes neuronales para formar cúmulos.

Thagard[1] tiene un enfoque distinto al de Langley y Simon. Recurre a las redes neuronales para formar cúmulos[2] de postulados de una teoría (Thagard [1992]). Los postulados en un cúmulo son coherentes entre sí. De esa manera explica la coherencia de las teorías y conforme se incorporan más leyes se modifican los cúmulos, con lo que explica los cambios conceptuales en las revoluciones científicas.

1.2 Descubrimiento científico como resolución de problemas

La hipótesis planteada por Langley, Simon, Bradshaw, y Zytkow [1987] es que los mecanismos del descubrimiento científico no son exclusivos de esa actividad, sino que son casos de mecanismos más generales de resolución de problemas. En su trabajo presentan varios programas capaces de hacer descubrimientos científicos no triviales.

Los programas hacen emplean métodos humanos de resolución de problemas, en particular el método de *búsqueda mediante selección heurística*. En esta tesis se entiende por *heurística* la ideación de hipótesis y teorías científicas, así como la resolución de problemas mediante tanteos sucesivos. Este enfoque es acorde con la propuesta de Newell y Simon [1972, 1963] .

Langley y sus colaboradores, como ya se mencionó, proponen que el descubrimiento científico es un caso particular de la resolución de problemas. Esta hipótesis es atractiva porque satisface el deseo de parsimonia que se espera de una teoría científica, restando importancia a la idea de que el descubrimiento científico es, en alto grado, un ejercicio de cierta facultad exclusiva de los humanos, dando pie a la posibilidad de una automatización.

Un ejemplo de una búsqueda por medio de heurística es la resolución del rompecabezas de los misioneros y los caníbales, que consiste en:

Estado inicial: 3 misioneros y 3 caníbales se encuentran a la orilla de un río que desean cruzar.

Restricciones:

[1] Quien acuñó el término *filosofía computacional de la ciencia* (Thagard [1988]).

[2] Cúmulo del latín *cumulus*, amontonamiento. En inglés se llaman clusters. En computación, particularmente en el área de reconocimiento de patrones, es importante formar cúmulos de objetos con características semejantes.

1. Se cuenta con una canoa con cupo para 2 personas, y es la única forma de cruzar el río.

2. Los caníbales nunca deben superar en número a los misioneros.

Objetivo: Los 6 deben cruzar el río sin violar las restricciones.

Para resolver el problema, primero hay que encontar una representación adecuada. Debido a su planteamiento, se necesita una estructura con información del estado que mantiene la distribución de misioneros y caníbales, tanto en la canoa como en ambas orillas del río. Un vector con dos vectores de dos números enteros cada uno, $((M_0, C_0), (M_1, C_1))$, donde las M_i representan el número de misioneros, y las C_i al número de caníbales de cada lado i del río, así el estado inicial se representa con $((3,3),(0,0))$, mientras que el estado final se representa por $((0,0),(3,3))$.

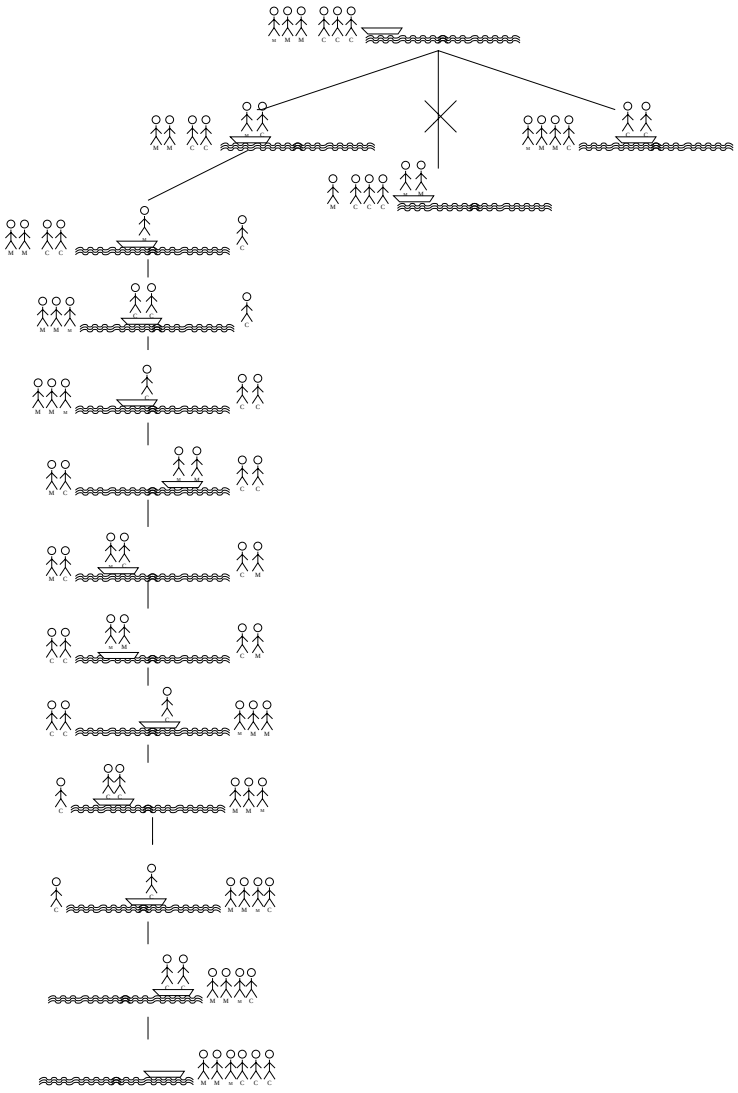
El espacio de solución del problema son todos los estados posibles del problema: $\{((3,3),(0,0)),\ ((3,2),(0,1)),\ ((3,1),(0,2)),\ ((3,0),(0,3)),\ ((2,3),(1,0)),\ ((2,2),(1,1)),\ ((2,1),(1,2)),\ ((2,0),(1,3)),\ ((1,3),(2,0)),\ ((1,2),(2,1)),\ ((1,1),(2,2)),\ ((1,0),(2,3)),\ ((0,3),(3,0)),\ ((0,2),(3,1)),\ ((0,1),(3,2)),\ ((0,0),(3,3))\}$.

Sin embargo, no todos los estados cumplen con la segunda restricción que prohíbe que los caníbales superen en número a los misioneros.

Las válidas son: $\{((3,3),(0,0)),\ ((3,2),(0,1)),\ ((3,1),(0,2)),\ ((3,0),(0,3)),\ ((2,2),(1,1)),\ ((1,1),(2,2)),\ ((0,3),(3,0)),\ ((0,2),(3,1)),\ ((0,1),(3,2)),\ ((0,0),(3,3))\}$

Para encontrar la solución, se puede construir un árbol de búsqueda partiendo del estado inicial, seguido de los estados que permiten las restricciones de que el número de caníbales no debe superar al número de misioneros y que se permite no más de dos pasajeros en la canoa.

De $((3,3),(0,0))$ pueden seguir $((3,2),(0,1))$, $((3,1),(0,2))$, o $((2,2),(1,1))$, después se toma a cualquiera de los estados resultantes como inicial y se aplican las restricciones para producir la siguiente generación, hasta construir todo el árbol. Los resultados son todas las rutas que partan del estado inicial $((3,3),(0,0))$ y que tengan como estado final $((0,0),(3,3))$. El siguiente diagrama muestra una solución:



Si se hiciese una búsqueda exhaustiva, el árbol tendría $16! = 20\,922\,789\,888\,000$ nodos. Empero, analizando detenidamente las restricciones y el objetivo, puede encontrarse un procedimiento más eficiente que encuentra una solución recorriendo una docena de estados.

Ello es posible gracias a la búsqueda con selección heurística, aplicando reglas al tanteo[3] para reducir el espacio de búsqueda. Primero se generan únicamente los nodos que cumplen con las restricciones, después se recorren solo los caminos que aproximan al objetivo. Esto último no siempre es obvio, en uno de los pasos puede observarse que se requiere regresar la canoa con dos ocupantes, cuando podría pensarse que lo óptimo es que siempre regrese con uno. Los métodos heurísticos no son infalibles, por lo que es posible que no se encuentre el resultado.

Algunos autores como Simon, clasifican las *heurísticas* en *fuertes* y *dé-*

[3]En inglés se dice: rules of the thumb.

biles,[4] dependiendo del grado en que las reglas al tanteo sean capaces de resolver el problema. Cuando se tiene la garantía de que siempre encuentran las soluciones, si es que existen, se dice que la heurística es fuerte, en este caso se le llama *algoritmo*[5]. En cambio, si no reducen adecuadamente el espacio de búsqueda y no siempre encuentran las soluciones posibles, o se pierden entre aproximaciones relativas sin converger a la solución, se dice que la heurística es débil. Para distinguir de los algoritmos a las heurísticas que no garantizan solución se les llama a estas últimas métodos heurísticos o simplemente heurísticas.

Desafortunadamente, no existe un procedimiento general para encontrar heurísticas y menos aún algoritmos.[6] Sin embargo, hay varias familias de heurísticas de las que se puede partir para construir un procedimiento que resuelva el problema. Un buen inicio es la búsqueda con fuerza bruta, generando y recorriendo el árbol de búsqueda de manera ordenada. Existen dos formas principales de generarlo: *en profundidad*[7] y *a lo ancho*[8].[9]

No sólo es importante del orden de recorrido, sino también la estrategia con que se generan los nodos descendientes, también la manera de abordar el problema. Algunas alternativas son:

Divide y conquista: Se divide al problema en subproblemas más sencillos.

Trabajar hacia atrás: Se busca un camino del objetivo al estado inicial.

Procedimientos de tipo voraz: [10] En cada paso se acercan a la solución.[11]

Entre más información se tiene del problema, es posible diseñar un método más especializado con una *heurística fuerte*, incluso podría llegarse a garantizar la solución, es decir un algoritmo.

[4] También se llaman duras y suaves.

[5] Algunos autores consideran que cualquier procedimiento que pueda implementarse en una máquina de Turing, es un algoritmo. En esta tesis me apegaré a la definición de Herbert Simon, que también es consistente con las dadas en Knuth [1973] y Trakhtenbrot [1980].

[6] Aunque existen algunos métodos específicos para generar algoritmos de una clase particular de problemas.

[7] Depth First en inglés.

[8] Breadth First en inglés.

[9] Existen otras, por ejemplo Streamed Depth First, que es una búsqueda en profundidad que también genera algunos nodos a lo ancho

[10] Greedy en inglés.

[11] Un ejemplo de una búsqueda heurística del tipo voraz, es el calcular el cambio con el mínimo de monedas. El procedimiento consiste en primero dividir la cantidad entre la denominación de la moneda mayor, y el resto, dividirlo entre la siguiente denominación, hasta tener la cantidad completa en monedas.

1.3 Inducción

La mayoría de los programas que reproducen descubrimientos históricos, presentados en Langley *et al.* [1987], son dirigidos por datos, esto es, son programas que descubren leyes por medio de inducción. En ese sentido, por *inducción* se entiende un proceso inferecial que parte de un conjunto finito de hechos, o un cuerpo de datos, a una generalización universalmente cuantificada, es decir, una ley.[12]

Entre esos programas están BACON.1 , y GLAUBER. Es este último el que se analizará detalladamente en esta tesis.

BACON.1 reproduce el descubrimiento de la tercera ley de Kepler, a partir de las observaciones hechas por Tycho Brahe respecto al período y diámetro de las órbitas planetarias. Este método se conoce como inducción baconiana, el cual, como se ejemplifica en el capítulo 3 de Langley *et al.* [1987], simula otros descubrimientos como las leyes de los gases de Boyle.

BACON.1 busca términos (constantes) teóricos, aplicado a los datos (idealizados) que Tycho Brahe dio a Johanes Kepler, y a partir del uso de la inducción baconiana en tres pasos encuentra la constante teórica y por lo tanto la ley descubierta.

	Datos		Termino 1	Termino 2	Termino 3
Planeta	Distancia(D)	Periodo(P)	D/P	D^2/P	D^3/P^2
A	1	1	1	1	1
B	4	8	1/2	2	1
C	9	27	1/3	3	1

Con lo que se obtiene $D^3/P^2 = 1$ que es lo mismo que $D^3 = 1 \cdot P^2$.

La leyes descubiertas por GLAUBER y BACON.1 son descriptivas, es decir, permiten predecir los resultados de los experimentos, pero no brindan una explicación causal del fenómeno observado.

Los autores plantean que la distinción entre descripción y explicación no es dicotómica sino graduada. En el caso de la tercera ley de Kepler, descubierta por BACON.1, que en la actualidad se considera descriptiva (no por Kepler), contiene algunos términos teóricos. La aparición de términos teóricos la vuelve menos descriptiva y un poco explicativa. En ese sentido, la ley de la gravitación universal de Newton $M_1\, a_1 = g\frac{M_1\, M_2}{d^2}$, donde M_1 y M_2 son las masas de los cuerpos, d la distancia, a_1 la aceleración del cuerpo 1 y g la constante gravitacional, tiene más términos teóricos, y la tercera ley de Kepler es un caso particular de ésta, por lo que la explica.

[12] Ver Langley *et al.* [1987] p.14

En contraparte al descubrimiento dirigido por datos, existe el descubrimiento dirigido por la teoría. Una vez que se han encontrado las leyes de una teoría, expresadas como fórmulas de un sistema con un aparato deductivo, tales fórmulas generan un espacio de búsqueda. Éstas representan alguna relación de los objetos en el mundo real de donde se toman los datos y tienen un significado. Un problema también puede representarse con el lenguaje del sistema formal, junto con las leyes encontradas, para procesarse buscando regularidades con los métodos de solución de problemas, a fin de encontrar una solución general o una nueva ley que corresponda con una relación dentro del mundo real. A esta forma se le llama *descubrimiento dirigido por la teoría*. En Langley *et al.* [1987] también se presentan algunos ejemplos de descubrimientos dirigidos por la teoría.

Una limitación de GLAUBER es que aunque descubre leyes para la mayoría de los casos fracasa con las excepciones. Es incapaz de descubrir que los ácidos débiles reaccionan con ácidos fuertes. El enfoque del método SURREALIST que desarrollé para esta tesis, representa la información en el álgebra relacional, lo que permite generar leyes correctamente cuantificadas.

Para que distinga entre $\forall x \cdot \exists y \cdot \varphi(x, y)$

Capítulo 2

El programa GLAUBER

En el siglo XVII, *Johann Rudolph Glauber* descubrió que los ácidos reaccionan con álcalis para formar sales. Esto fue posible gracias a que los químicos de su época ya habían clasificado muchas substancias basándose en las cualidades que podían observar, tales como *sabor* y *textura*, así como las *interacciones entre substancias*. Langley, Simon, Bradshaw, y Zytkow [1987] encontraron en este hecho histórico un excelente ejemplo de inducción de leyes cualitativas con el cual corroborar su afirmación de que *"descubrimiento científico equivale a resolución de problemas"* (ver 1.1 y 1.3). Presentan el programa GLAUBER[1], que simula exitosamente un descubrimiento dirigido por datos, es decir, de carácter inductivo. GLAUBER es un procedimiento heurístico que recibe como entrada las cualidades de las substancias y la relación de reacción entre substancias:

```
(Has-Quality Object {NaOH} Tastes {Bitter})
(Has-Quality Object {KOH} Tastes {Bitter})
(Has-Quality Object {HCl} Tastes {Sour})
        ⋮          ⋮          ⋮
(Reacts Inputs {HCl KOH} Outputs {KCl})
(Reacts Inputs {HNO3 KOH} Outputs {KNO3})
        ⋮          ⋮          ⋮
```

Con la información sobre las cualidades de las substancias forma cúmulos y los etiqueta. Para encontrar un patrón común en la información de las reacciones sustituye en la información de las reacciones, las substancias por la etiqueta correspondiente al cúmulo al que pertenecen. Con la cualidad común a las substancias de cada cúmulo forma las clases correspondientes. En

[1] Aunque llamaron GLAUBER al programa en honor al químico que hizo el descubrimiento, no fue el único en dar aportaciones para entender el fenómeno. Por la misma época *Jan Bapista van Helmont* y *Otto Techenius* descubrieron que los ácidos reaccionan con metales para formar sales, por lo que se habla de los *álcalis* y los *metales* como *bases*, y qué los *ácidos reaccionan con bases para formar sales*.

el ejemplo que presentan los autores, mismo que se utilizará en este capítulo para explicar el procedimiento, se encuentran las clases *ácidos*, *álcalis* y *sales*, como resultado de partir al conjunto de substancias en clases de equivalencia mediante la cualidad sabor (ácido, amargo o salado), por ser la que las distingue. Para generalizar la ley se determina la cuantificación adecuada. Johann Rudolph Glauber descubrió que *todos los ácidos* reaccionan con *todos los álcalis* formando *alguna sal*. Alimentado con datos que se conocían en la época del descubrimiento, el programa GLAUBER produce la siguiente salida:

∀alkali ∀acid ∃salt (Reacts Inputs {acid, alkali} Outputs {salt})
∀salt (Has-Quality Object {salt} Tastes {Salty})
∀acid (Has-Quality Object {acid} Tastes {Sour})
∀alkali (Has-Quality Object {alkali} Tastes {Bitter})

Donde la primera fórmula es la generalización de los hechos `Reacts`[2] y las tres últimas reglas describen como se forman las clases descubiertas.

2.1 El procedimiento GLAUBER

A continuación se describe el procedimiento GLAUBER como se presenta en Langley, Simon, Bradshaw, y Zytkow [1987]. Los autores no describen todos los detalles, salvo lo citado. Lo que se muestra es mi interpretación de lo expuesto por ellos.

2.1.1 Representación de la información

Como ya se mencionó existen dos tipos de predicados, `Has-Quality` y `Reacts` los cuales cuentan con la siguiente estructura[3]:

$$\texttt{Reacts}\begin{cases}\texttt{Inputs} & \{substance^{+}\}\\ \otimes & \\ \texttt{Outputs} & \{substance^{+}\}\end{cases}\qquad \texttt{Has-Quality}\begin{cases}\texttt{Object} & substance\\ \otimes & \\ \texttt{Quality} & quality\\ \otimes & \\ \texttt{Value} & value\end{cases}$$

Vistas como los predicados:

$$\text{Reacts}\left(\text{Inputs}\left(substance^{+}\right), \text{Outputs}\left(substance^{+}\right)\right)$$

$$\text{Has-Quality}\left(\text{Object}\left(substance\right), quality\left(value\right)\right)$$

Estos se representan con las listas:[4]

[2] ∀ se lee para todo y ∃ se lee existe.

[3] El símbolo ⊗ es un producto, y el signo α^{-} significa una o más ocurrencias de α.

[4] Aunque la sintaxis de GLAUBER podría ser igual en Lisp, considerando que en muchos dialectos pueden usarse '{','}' o '[',']' en lugar de '(',')', GLAUBER está escrito en PRISM, un sistema de producción que a su vez está escrito en Lisp.

(Reacts Inputs {*substancias*} Outputs {*substancias*})
(Has-Quality Object {*substancia*} *cualidad* {*valor*})

La sintaxis está fuertemente influida por el uso de listas al estilo de Lisp[5]. El ejemplo que presentan Langley *et al.* [1987] es el siguiente:

(Reacts Inputs {HCl NaOH} Outputs {NaCl})
(Reacts Inputs {HCl KOH} Outputs {KCl})
(Reacts Inputs {HNO_3 NaOH} Outputs {$NaNO_3$})
(Reacts Inputs {HNO_3 KOH} Outputs {KNO_3})
(Has-Quality Object {HCl} *Tastes* {*Sour*})
(Has-Quality Object {HNO_3} *Tastes* {*Sour*})
(Has-Quality Object {NaOH} *Tastes* {*Bitter*})
(Has-Quality Object {KOH} *Tastes* {*Bitter*})
(Has-Quality Object {NaCl} *Tastes* {*Salty*})
(Has-Quality Object {$NaNO_3$} *Tastes* {*Salty*})
(Has-Quality Object {KCl} *Tastes* {*Salty*})
(Has-Quality Object {KNO_3} *Tastes* {*Salty*})

Cabe aclarar que aunque las substancias en *inputs* se representan con listas, son tratadas como conjuntos. A su vez las substancias, si bien denominadas por su fórmula, son tratadas como símbolos, no representan la estructura química, simplemente son nombres dados a las substancias, tal y como podrían haberse empleado los nombres *sosa*, *potasa*, *ácido muriático*[6] o *agua régia*[7] en lugar de NaOH, KOH, HCl o HNO_3. Las cadenas empleadas para los nombres no tienen un significado dentro del programa. Tampoco hay forma de definir sinónimos, es decir, para GLAUBER el símbolo *sosa* no es equivalente al símbolo NaOH, aunque la modificación para definir distintos símbolos como equivalentes no representa mayor dificultad.[8]

[5]Lisp: List Processing Language. Es un lenguaje ampliamente utilizado en Inteligencia Artificial. En Lisp todo es una expresión simbólica (Exp-S) que se define inductivamente como sigue: a) un símbolo es una Exp-S, b) ( ) es una Exp-S, c) $(e_1 \cdot e_2)$ es una Exp-S, si e_1 y e_2 son Exp-S, d) solo es una Exp-S si está definido con las reglas anteriores. Un "azúcar sintáctico" para las Exp-S, es la notación de listas. La expresión ((a·(b·()))·(c·())) se escribe ((a b) c). Para mayor información sobre Lisp véase a McCarthy [1960] y Henderson [1980].

[6]Del lat. *murĭa*, salmuera, ácido clorhídrico.

[7]De *aqua regia* por atacar al oro, considerado antiguamente el rey de los metales. En este trabajo se considera sinónimo del ácido nítrico, aunque en realidad es una mezcla 1 : 3 de ácidos nítrico y clorhídrico.

[8]Hacer un análisis léxico que devuelva el mismo valor para los sinónimos es una modificación relativamente sencilla. En cambio, sería una extensión interesante el proveer al programa de lógica para descubrir que dos substancias resultan ser la misma con distintos nombres, por ejemplo detectando sinónimos cuando exista manera de detectarlos por sus atributos, pero no se abordará en esta tesis.

2.1.2 Procedimientos `Form-Class` y `Determine-Quantifier`

El procedimiento GLAUBER consiste en los procedimientos `Form-Class` y `Determine-Quantifier` que transforman la información, hasta encontrar una ley. En el ejemplo se obtendrá:

(∀ *Acid* ∀ *Alkali* ∃ *Salt* (Reacts Inputs {*Acid Alkali*} Outputs {*Salt*}))

Se muestra el procedimiento comentando la explicación dada por sus autores, usando la tabla 6.1 en la página 204 de Langley *et al.* [1987]. Asimismo, se intercaló la explicación del procedimiento aplicado para pasar de un estado a otro por lo que fue necesario agregar los estados marcados con ‡.
El estado inicial contiene únicamente los hechos conocidos, esto es, los datos de entrada pasados al programa:

Estado inicial, S1
(Reacts Inputs {HCl,NaOH} Outputs {NaCl})
(Reacts Inputs {HCl,KOH} Outputs {KCl})
(Reacts Inputs {HNO_3,NaOH} Outputs {$NaNO_3$})
(Reacts Inputs {HNO_3,KOH} Outputs {KNO_3})
(Has-Quality Object {HCl} Tastes {Sour})
(Has-Quality Object {HNO_3} Tastes {Sour})
(Has-Quality Object {NaCl} Tastes {Salty})
(Has-Quality Object {$NaNO_3$} Tastes {Salty})
(Has-Quality Object {KCl} Tastes {Salty})
(Has-Quality Object {KNO_3} Tastes {Salty})
(Has-Quality Object {NaOH} Tastes {Bitter})
(Has-Quality Object {KOH} Tastes {Bitter})

> *Dados los doce hechos como entradas,* GLAUBER *comienza examinando varias tripletas (predicado, atributo, valor) y determina cuales ocurren en el mayor número de hechos. Éste nota que los símbolos* HCl, HNO_3, *son argumentos de cada entrada* inputs *para dos hechos donde el predicado es* reacts. *Similarmente, cada uno de los símbolos* sour *y* bitter *ocurren como argumento de la entrada taste en dos hechos has-quality. Sin embargo, el símbolo con mayor puntaje es* salty, *que ocurre en cuatro hechos* has-quality *como el valor para* taste.[9]
>
> *Esta tripleta es seleccionada, y esos cuatro hechos se reemplazan por la ley* (Has-Quality Object {salt} Tastes {Salty}), *que*

[9][Nota de los autores] *Si hubiésemos representado la información taste usando los predicados sour, bitter, y salty,* GLAUBER *no podría haber formulado esta clase, dado que la heurística del sistema busca valores compartidos, los sabores de las substancias deben almacenarse como valores en lugar de predicados.*

tiene la misma forma que las proposiciones originales en las cuales los valores diferentes en la entrada object son reemplazados por el nombre de la clase "salt". Además, las substancias NaCl, KCl, $NaNO_3$, *y* KNO_3 *se almacenan como elementos de la nueva clase.*

Langley, Simon, Bradshaw, y Zytkow [1987] p.205.

Estado S2‡
SALTS: {NaCl,KCl,$NaNO_3$,KNO_3}
(Reacts Inputs {HCl,NaOH} Outputs {salt})
(Reacts Inputs {HCl,KOH} Outputs {salt})
(Reacts Inputs {HNO_3,NaOH} Outputs {salt})
(Reacts Inputs {HNO_3,KOH} Outputs {salt})
(Has-Quality Object {HCl} Tastes {Sour})
(Has-Quality Object {HNO_3} Tastes {Sour})
(Has-Quality Object {NaOH} Tastes {Bitter})
(Has-Quality Object {KOH} Tastes {Bitter})
(Has-Quality Object {salt} Tastes {Salty})

Aparte de proponer esta ley, el operador Form-Class *genera cuatro hipótesis adicionales sustituyendo el símbolo "salt" por los elementos de esta clase dentro de otros hechos. De aquí que, los hechos* (Reacts Inputs {HCl, NaOH} Outputs {NaCl}) *y* (Reacts Inputs {HCl, KOH} Outputs {KCl}), *son reemplazados por* (Reacts Inputs {HCl, NaOH} Outputs {salt}) *y* (Reacts Inputs {HCl, KOH} Outputs {salt}). *Similarmente, los hechos* (Reacts Inputs {HNO_3, NaOH} Outputs {$NaNO_3$}) *y* (Reacts Inputs {HNO_3,KOH} Outputs {KNO_3}), *se reemplazan por* (Reacts Inputs {HNO_3, NaOH} Outputs {salt}) *y* (Reacts Inputs {HNO_3, KOH} Outputs {salt}). *Aunque se tiene la garantía de que la primera de estas leyes está cuantificada universalmente por la manera en que fue definida la clase "salt", la generalidad de las otras leyes debe determinarse empíricamente. Por ejemplo, si la ley* (Reacts Inputs {HCl,NaOH} Outputs {salt}) *fuese cuantificada universalmente sobre la clase de las sales, entonces serían predichos cuatro hechos. Dado que solo se ha observado una de esas predicciones,* GLAUBER *emplea un cuantificador existencial en lugar de uno universal. Se toma la misma decisión con las otras leyes formadas por substitución; lo que nos lleva a las leyes y hechos que se muestran en la segunda sección de la tabla 6.1*

Langley, Simon, Bradshaw, y Zytkow [1987] p.206.

`Determine-Quantifier` es el procedimiento que calcula el cuantificador adecuado. Así se llega a un estado donde se conoce el cuantificador en el predicado `Reacts`, del predicado `Has-Quality` correspondiente y la extensión de la clase.

`Form-Class` y `Determine-Quantifier` llevan al estado S3
SALTS:{KNO_3,KCl,$NaNO_3$,NaCl}
∃ salt (Reacts Inputs {HCl,NaOH} Outputs {salt})
∃ salt (Reacts Inputs {HCl,KOH} Outputs {salt})
∃ salt (Reacts Inputs {HNO_3,NaOH} Outputs {salt})
∃ salt (Reacts Inputs {HNO_3,KOH} Outputs {salt})
∀ salt (Has-Quality Object {salt} Tastes {Salty})
(Has-Quality Object {HCl} Tastes {Sour})
(Has-Quality Object {HNO_3} Tastes {Sour})
(Has-Quality Object {NaOH} Tastes {Bitter})
(Has-Quality Object {KOH} Tastes {Bitter})

Mientras existan nombres de substancias en los predicados `Reacts`, se vuelven a aplicar los procedimientos `Form-Class` y `Determine-Quantifier`.

En este caso, al aplicar `Form-Class`, se encuentran los conjuntos de predicados `Has-Quality` con las cualidades *Tastes* {*Bitter*} y *Tastes* {*Sour*}. Ambas abarcan todos los hechos reacts y ambas contienen el mismo número de substancias, por lo que puede seleccionarse cualquiera de las dos. Tomando *Tastes* {*Sour*} se obtiene:

Estado S4‡
SALTS: {NaCl,KCl,$NaNO_3$,KNO_3}
ACIDS: {HCl,HNO_3}
(∃ salt (Reacts Inputs {acid, NaOH} Outputs {salt}))
(∃ salt (Reacts Inputs {acid, KOH} Outputs {salt}))
(∃ salt (Reacts Inputs {acid, NaOH} Outputs {salt}))
(∃ salt (Reacts Inputs {acid, KOH} Outputs {salt}))
(Has-Quality Object {NaOH} Tastes {Bitter})
(Has-Quality Object {KOH} Tastes {Bitter})
(∀ salt (Has-Quality Object {salt} Tastes {Salty}))
(Has-Quality Object {acid} Tastes {Sour})

Ahora se aplica el procedimiento `Determine-Quantifier`, prediciendo todos los predicados `Reacts`, por lo que el cuantificador correspondiente es el universal, obteniéndose:

Estado S5
SALTS:{KNO_3,KCl,$NaNO_3$,NaCl}
ACIDS:{HCl,HNO_3}
∀ acid ∃ salt (Reacts Inputs {acid,NaOH} Outputs {salt})
∀ acid ∃ salt (Reacts Inputs {acid,KOH} Outputs {salt})
∀ salt (Has-Quality Object {salt} Tastes {Salty})
∀ acid (Has-Quality Object {acid} Tastes {Sour})
(Has-Quality Object {NaOH} Tastes {Bitter})
(Has-Quality Object {KOH} Tastes {Bitter})

El estado actual todavía tiene nombres de substancias en los predicados `Reacts`, por lo que se vuelven a aplicar los procedimientos `Form-Class` y `Determine-Quantifier`. Al aplicar `Form-Class` solo hay una clase candidata formada por los predicados `Has-Quality` con los valores *Tastes* {*Sour*}. Lo anterior nos lleva a:

Estado S6‡
SALTS:{KNO_3,KCl,$NaNO_3$,NaCl}
ACIDS:{HCl,HNO_3}
ALKALIS: {NaOH,KOH}
(∀ acid ∃ salt (Reacts Inputs {acid,alkali} Outputs {salt}))
(∀ acid ∃ salt (Reacts Inputs {acid,alkali} Outputs {salt}))
(∀ salt (Has-Quality Object {salt} Tastes {Salty}))
(∀ acid (Has-Quality Object {acid} Tastes {Sour}))
(Has-Quality Object {alkali} Tastes {Bitter})

Al generar los predicados `Reacts` con todos los miembros de la clase, se obtienen los dos ya conocidos, por lo que el cuantificador correspondiente es el universal, quedando:

Estado final S7
SALTS:{KNO_3,KCl,$NaNO_3$,NaCl}
ACIDS:{HCl,HNO_3}
ALKALIS:{NaOH,KOH}
∀ alkali ∀ acid ∃ salt (Reacts Inputs {acid,alkali} Outputs {salt})
∀ salt (Has-Quality Object {salt} Tastes {Salty})
∀ acid (Has-Quality Object {acid} Tastes {Sour})
∀ alkali(Has-Quality Object {alkali} Tastes {Bitter})

No hay más nombres de substancias en los predicados `Reacts`. GLAUBER terminó con éxito. Todos los nombres de substancias fueron substituidos por variables que toman valores de las clases encontradas. Las reglas descubiertas generan a los ejemplares sobre los que se aplicó el procedimiento inductivo para obtenerlas y predicen los resultados de nuevos experimentos.

2.1.3 Resumen del procedimiento GLAUBER

El procedimiento GLAUBER se resume a continuación:

Procedimiento: GLAUBER

Entrada: Los conjuntos de predicados `Reacts` y `Has-Quality`.

Salida: Si tiene éxito, un conjunto de clases descritas con las cualidades en los predicados `Has-Quality` y generaliza los predicados `Reacts` con variables que toman variables de las clases descubiertas.

G1. **Si** existen nombres de substancias en los predicados `Reacts`,

entonces continúa con el paso G2,

en caso contrario termina.

G2. Aplica `Form-Class` al conjunto de entrada.

G3. Aplica `Determine-Quantifier` al resultado del paso anterior.

G4. Continúa con el paso G1.

Procedimiento: `Form-Class`

Entrada: Los conjuntos de predicados `Reacts` y `Has-Quality`

Salida: Una nueva clase

F1. Cuenta los predicados `Has-Quality` que tienen el mismo valor en las entradas *Quality*`{`*Value*`}`.

F2. Selecciona el que tenga el mayor numero de ocurrencias y que las substancias aparezcan en los predicados `Reacts`.

F3. Crea un nombre para la clase.

F4. Reemplaza todos los predicados `Has-Quality` con las entradas *Quality*`{`*Value*`}` de la clase seleccionada en el paso F2 por un nuevo predicado (`Has-Quality {`*nombre*`}` *Quality*`{`*Value*`}`) donde *nombre* es el generado en el paso F3 y los valores en las entradas *Quality*`{`*Value*`}` son los comunes a los predicados seleccionados.

F5. En cada predicado `Reacts`, se reemplaza el nombre de la substancia contenida en la clase encontrada en el paso F2 por el nombre generado en el paso F3.

Procedimiento: `Determine-Quantifier`

Entrada: Los conjuntos de predicados `Reacts` y `Has-Quality` con los predicados generados por `Form-Class`

Salida: Agrega a los predicados las regla para formar la clase, y cuantifica adecuadamente los predicados `Reacts`

D1. Cuantifica universalmente la regla para definir la clase.

D2. Genera todos los predicados `Reacts` reemplazando el nombre de la clase con todos los objetos en la clase nueva.

D3. Prueba si todos los predicados generados están contenidos en el conjunto original de predicados `Reacts`.

Si todos los predicados están contenidos en el conjunto original

Entonces cuantifica universalmente.

En otro caso cuantifica existencialmente.

2.2 Limitaciones de GLAUBER

A continuación se reproduce lo que los autores mencionan sobre las limitaciones de GLAUBER (ver Langley *et al.* [1987], páginas 207-209):

1. *El primer problema gira en torno al tratamiento que el programa da a los cuantificadores y su orden. La expresión* $(\forall x\ \exists y\ P(x,y))$ *no es equivalente a* $(\exists y\ \forall x\ P(x,y))$*; la segunda fórmula es más específica que la primera, por lo que hace una afirmación más fuerte.*

   Aunque GLAUBER *puede llegar a leyes de la segunda forma, esto ocurre solo si sucede que se definen las clases en cierto orden; el sistema no encuentra leyes maximalmente especificas en todos los casos que debería.*

2. *Un segundo problema relacionado con el orden de los cuantificadores concierne a las leyes complementarias, tales como* $(\forall x\,\exists y\,P(x,y))$ *y* $(\forall y\,\exists x\,P(x,y))$*. Hemos visto que tales leyes se consideran cuando dos clases se definen en el mismo paso; sin embargo, hay otros casos*

en que a uno le gustaría que esto ocurriera. Por ejemplo, la tabla 6.1[10] *resume como* GLAUBER *descubre la ley* $\forall alkali\, \forall acid\, \exists salt\, (reacts\ inputs\{acid\ alkali\}\ ouputs\{salt\})$. *Esto establece que todos los ácidos reaccionan con todos los álcalis para formar alguna sal. Sin embargo, los datos originales también soportan la ley complementaria,* $\forall salt\, \exists alkali\, \exists acid\, (reacts\ inputs\{acid\ alkali\}\ ouputs\{salt\})$, *que establece que todas las sales son producto de una reacción entre algún ácido y algún álcali.* GLAUBER *habría encontrado esta ley si su heurística hubiera tomado la primera alternativa*[11] *(primero definir ácidos, después álcalis, y finalmente las sales), pero la versión existente nunca podría haber encontrado ambas leyes en la misma ejecución.*

3. *Otras limitaciones se relacionan con la función de evaluación para dirigir la búsqueda en el espacio de clases y leyes. La versión actual itera a través del conjunto de tripletas (predicado, atributo, valor) y selecciona la tripleta que ocurre en el mayor número de hechos. Esto lleva a* GLAUBER *a preferir las clases mayores sobre las menores, que a su vez llevan a leyes con mayor generalidad (en el sentido de que ellos cubren más hechos observados). Sin embargo, recordemos que una vez que* GLAUBER *define una clase nueva con fundamento en alguna ley, procede a crear leyes adicionales sustituyendo la clase por sus miembros en otros hechos. Esto sugiere una definición más amplia de generalidad, incluyendo todos los hechos predichos por cualquier ley que comprenda a la clase nueva. Este análisis nos lleva a dos métodos para preferir una ley sobre otra. El enfoque más obvio comprende el computar el porcentaje de predicciones que verdaderamente es sustentado por las observaciones; a éste le llamaremos el* poder predictivo *de la clase y sus leyes asociadas. El segundo método consiste en computar el número total de hechos predichos por la clase y sus leyes relacionadas; llamaremos a éste el* potencial predictivo *de la clase.*

   Obviamente, una ley que predice pocas reacciones observadas y predice muchas no observadas es indeseable; ello sugiere que el poder predictivo debe usarse para eliminar grosso mo-

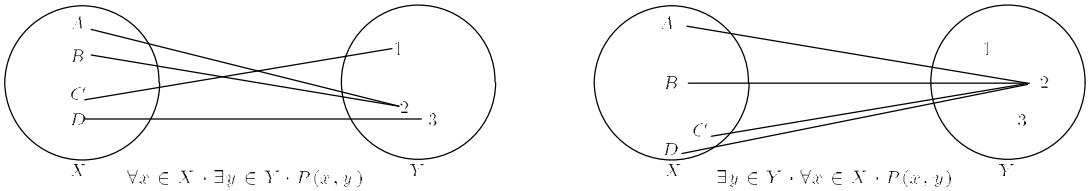
[10] Es la tabla que se usó en la sección 2.1.2 en la página 16 de esta tesis para explicar los procedimientos `Form-Class` y `Determine-Quantifier`.

[11] Los autores se refieren a que en `Form-Class` seleccionara primero las clases con menor tamaño.

> *do las clases inaceptables. Sin embargo, cuando los puntajes en esta dimensión son aproximadamente iguales, deben preferirse los conjuntos de leyes con mayor potencial predictivo, dado que ellas llevan a más predicciones que, si son satisfechas, llevarán a un incremento en dicho potencial. Una manera de implementar este esquema es que* GLAUBER *tuviera que generar las clases potenciales y sus leyes asociadas, para poder determinar su poder y potencial predictivo. Entonces, el sistema tendría que considerar si esas leyes deben calificarse existencialmente o universalmente para poder maximizar su puntaje. En otras palabras, el sistema debería aplicar el operador* `Form-Class` *en todas las formas posibles, y entonces aplicar el operador* `Determine-Quantifier` *en todas las formas posibles, para poder determinar la mejor ruta a seguir. Esto equivale a una vista adelantada*[12] *de dos pasos en el árbol de búsqueda, y ello involucraría un tiempo considerablemente mayor que el de la estrategia sencilla. Aunque los detalles de este esquema quedan por elaborarse, la idea básica de definir clases que explican la mayoría de los datos parece un enfoque verosímil.*
>
> *Sin embargo, para poder implementar esta estrategia, primero tendríamos que tratar con otras dos limitaciones del sistema actual. La distinción entre poder predictivo y potencial predictivo solo tiene sentido si uno puede probar predicciones, y la prueba de predicciones tiene sentido solo si tales predicciones pueden fallar. Esto significa que* GLAUBER *debe ser capaz de representar hechos negados u observaciones "falladas".*

Las limitaciones de GLAUBER reportadas por sus autores se deben principalmente a la manera en que se representa y procesa la información mediante listas. La primera limitación se debe a que el operador `Determine-Quantifier` compara los hechos predichos con los hechos dados y determina un cuantificador universal cuando los conjuntos son iguales, o un cuantificador existencial si los hechos dados son un subconjunto de los hechos predichos, así es imposible detectar casos como el mencionado por los autores y que corresponde a las figuras:

[12] El término que se usa en la literatura en inglés, así como el texto original es *look-ahead*.



El procedimiento `Determine-Quantifier` puede modificarse para que pueda determinar el cuantificador correcto cambiando la estrategia. En lugar de comparar el conjunto de hechos dados con el de las predicciones, puede contarse el tamaño del dominio y del rango. Si son del mismo tamaño la correspondencia es uno a uno y sobre, que puede expresarse mediante las cuantificaciones complementarias $\forall x \cdot \exists! y \cdot x \in X \wedge y \in Y \wedge P(x,y)$ y $\forall y \cdot \exists! x \cdot x \in X \wedge y \in Y \wedge P(x,y)$. En el caso en que el número de hechos observados corresponda a $|X| \times |Y|$, lo que es equivalente a comparar el conjunto de los hechos dados con el de las predicciones, puede deducirse que corresponde la cuantificación $\forall x \cdot \forall y \cdot x \in X \wedge y \in Y \wedge P(x,y)$, y en los otros casos corresponde la cuantificación existencial. Nuestra implementación se basa en la siguiente regla:

$$Cuantificador = \begin{cases} \forall & \text{si } \frac{|reacts|}{|reacts\{[x{:=}Clase] \div x \in Clase\}|} = |Clase| \\ \exists! & \text{si } \frac{|reacts|}{|reacts\{[x{:=}Clase] \div x \in Clase\}|} = 1 \\ \exists & \text{si } 1 < \frac{|reacts|}{|reacts\{[x{:=}Clase] \div x \in Clase\}|} < |Clase| \end{cases}$$

En la siguiente sección se dará una explicación más detallada y su implementación se describe en el apéndice A.

La segunda limitación se debe a que el procedimiento `Form-Class` de GLAUBER procesa una por una las cualidades para formar clases candidatas seleccionando la de mayor tamaño para substituir cada substancia de la clase en los predicados reacts por la clase a que pertenece. De esta forma se deja de lado una propiedad importante. Se trata de la relación que existe entre el conjunto de reactivos (*input*) y productos (*output*). En el ejemplo utilizado hay una correspondencia uno a uno entre las entradas y las salidas que corresponde a cualquiera de las cuantificaciones $\forall x \exists! y P(x,y)$ y $\forall y \exists! x P(x,y)$ donde x y y son conjuntos que al clasificar las substancias y formar clases pueden transformarse a tuplas, mismas que pertenecen a una relación. De aquí que la cuantificación anterior sea sobre relaciones y no sobre substancias.[13] Sin embargo la relación dada como entrada es construida con las clases descubiertas. La cuantificación de las substancias en la relación puede corresponder a cualquier combinación de cuantificadores. En el ejemplo x corresponde a una relación que es el producto cartesiano de los ácidos y los álcalis $ácidos \times álcalis$ que corresponde a la cuantificación $\forall ácido \forall álcali\ inputs(ácido, álcali)$. En

[13]Esto ya no es lógica de primer orden.

el capítulo 3 se verá como el procedimiento SURREALIST toma en cuenta lo anterior y así encuentra la cuantificación adecuada.

La última limitación relativa a la estrategia con que se hace la búsqueda. Al igual que las anteriores se debe a que las clases se forman una a la vez y tomando en cuenta para su formación a una sola cualidad. Lo anterior lleva a los autores a pensar en un criterio de maximalidad para mejorar su procedimiento. No aclaran si dicho criterio es de carácter estadístico con alguna medida de confiabilidad para la validez de la teoría descubierta. En este caso, a mi entender, las leyes descubiertas serían más bien de carácter descriptivo, las excepciones al ser mínimas no cuentan para la formulación de la teoría. A fin de cuentas, las leyes predicen confiablemente los resultados de los experimentos. Este tipo de estrategias con criterios de maximalidad también es usado en programas como $ID3$ (ver Michalski, Bratko, y Kubat [1996]) que construye árboles de decisión para formar clases.

SURREALIST tiene otro enfoque. Pretende explicar todos los casos, busca en un solo paso todas las cualidades que parten al conjunto de los reactivos en clases de equivalencia y, aunque en el ejemplo solo tiene una substancia (en un caso más general puede tener más substancias), también se aplica a los productos.

Por supuesto que puede darse el caso de que las clases de equivalencia solo puedan formarse en un subconjunto propio de los hechos. Esto solo significa que los hechos restantes se rigen por otra(s) reglas que involucra(n) a otras clases. De esta forma puede obtenerse un resultado más fino. En los métodos que usan un criterio estadístico se desprecian las excepciones que sean minoría. No interesa el por qué.

Sin embargo, GLAUBER intenta encontrar leyes que cubran todos los datos. Por lo que entiendo, las modificaciones que esbozan los autores tienden a incorporar un criterio estadístico que a fin de cuentas desechará las excepciones que queden debajo del criterio de confiabilidad. Como acabamos de ver, el criterio de maximalidad mencionado por los autores no es necesario en SURREALIST porque éste busca reglas para todos los casos.

2.3 Una implementación de GLAUBER

En el apéndice A se muestra el código de la implementación de GLAUBER, que se explica en esta sección. Se describen los procedimientos considerando los detalles no descritos por los autores.

Para implementar GLAUBER fue necesario arreglar algunos detalles que no se mencionan en Langley *et al.* [1987]. El procedimiento `Form-Class` propone la mayor de las clases que se obtienen a partir de `Has-Quality`, pero es necesario que cada reacción en el conjunto `Reacts` contenga a una substancia

en la clase generada por `Form-Class`.[14] Cuando se viola esta condición puede arreglarse separando los predicados `Reacts` en tres grupos, los que tienen exactamente una substancia que pertenece a la clase propuesta, es decir los que son clasificados, los que no contienen ninguna substancia en la clase, y los que contienen más de una substancia en la clase. Los últimos dos casos no son clasificados por la clase y lo que puede hacerse es lo siguiente:

i. Si `Reacts` no tiene substancias en común con la clase propuesta $substs(Reacts) \cap clase = \varnothing$, entonces separa en otro conjunto al que se le aplicará GLAUBER por separado. Se obtendrán dos leyes, una ley L_1 para los que son clasificados, y la otra ley L_2 para los que no son clasificados con la clase pero que serán clasificados con otra clase porque son tratados como un caso independiente obteniéndose la segunda. Al terminar el proceso se juntan en la ley $L = L_1 \vee L_2$.

ii. Si `Reacts` tiene dos o más substancias en común con la clase propuesta $|substs(Reacts) \cap clase| > 1$ esto es, ninguna de las clases generadas por `Form-Class` clasifica a las substancias, puede intentarse creando nuevas clases con intersecciones entre las clases obtenidas anteriormente, y repetir el procedimiento de probar si es una clase viable, si vuelve a fallar, volver a generar nuevas clases con más intersecciones y así sucesivamente hasta que pueda hacerse la clasificación o se agoten las posibilidades. Si se encuentran intersecciones que formen clases que clasifiquen todas las substancias adecuadamente tendrán la forma $\{x \ni$ (Has-Quality Object $\{x\}$ $q_1\{v_1\}) \wedge$ (Has-Quality Object $\{x\}$ $q_2\{v_2\})\}$

En el procedimiento que se muestra a continuación no se implementó la solución de todos los casos por su complejidad[15]. De todos modos una ventaja del programa SURREALIST, implementado con el álgebra relacional, cuenta con una heurística más fuerte para formar clases. La prueba de las propiedades clasificatorias de la clase generada por `Form-Class` solo se incorporó para podar la búsqueda eliminando las que no clasifican, por lo que el comportamiento es como el GLAUBER que se presenta en Langley *et al.* [1987].[16]

[14] En el procedimiento que se da en la página 28, la validación se hace en los pasos 2(c)i y 2(c)ii.

[15] La complejidad se refiere a la eficiencia en función del tamaño de la entrada, en este caso se tienen todas las combinaciones posibles para formar intersecciones entre las clases, aunque solo es un cálculo del peor caso, haciendo experimentos es posible que la mayoría de las intersecciones sean vacías, y no tienen que hacerse muchas particiones del conjunto de reacciones.

[16] Los autores mencionan que hicieron otra versión del programa que busca de distinta

El siguiente procedimiento se basa en el programa original al que solo se le agregaron las validaciones mencionadas en el párrafo anterior, así como una modificación a `Determine-Quantifier` para que pueda encontrar el cuantificador de existencia única ∃! además del existencial y el universal:

1. GLAUBER recibe la información en listas de predicados `Reacts` y `Has-Quality`, con la forma:
   `(Has-Quality Object {HCl}` *Tastes* `{`*Sour*`})`
   `(Reacts Inputs {HCl, NaOH} Outputs {NaCl})`

2. Aplica `Form-Class` a los predicados `Has-Quality` para buscar la clase más adecuada.

   2a. Por cada cualidad y valor genera un conjunto definido por intensión como sigue:[17]
   $$clase\,(q_i, v_i) = \{x \mid \texttt{(Has-Quality Object } \{x\}\ q_i\ \{v_i\}\texttt{)}\}$$
   donde $clase(q_i, v_i)$ genera un nombre para la clase

   Por ejemplo, al aplicar *From-Class* a los hechos
   `(Has-Quality Object {HCl}` *Tastes* `{`*Sour*`})`
   `(Has-Quality Object {`HNO_3`}` *Tastes* `{`*Sour*`})`
   `(Has-Quality Object {NaOH}` *Tastes* `{`*Bitter*`})`
   `(Has-Quality Object {KOH}` *Tastes* `{`*Bitter*`})`
   `(Has-Quality Object {NaCl}` *Tastes* `{`*Salty*`})`
   `(Has-Quality Object {KCl}` *Tastes* `{`*Salty*`})`
   `(Has-Quality Object {`$NaNO_3$`}` *Tastes* `{`*Salty*`})`
   `(Has-Quality Object {`KNO_3`}` *Tastes* `{`*Salty*`})`

   del estado S1 se obtienen las siguientes clases candidatas (búsqueda a lo ancho):
   Tastes-Salty = {NaCl, KCl, $NaNO_3$, KNO_3}
   Tastes-Sour = {HCl, HNO_3}
   Tastes-Bitter = {NaOH, KOH}

   2b. Se selecciona la clase que contiene mayor número de elementos. En el ejemplo es *Tastes-Salty* que se devolverá junto con la regla que se cuantificará en el paso 3
   Tastes-Salty = $\{x \mid$ `(Has-Quality Object{`x`}` *Tastes*`{`*Salty*`})`

   2c. En cada predicado `Reacts` se sustituye a cada substancia $s \in C$ por la clase C. Deben cumplirse las condiciones:

---

forma las clases y afirman que es capaz de encontrar la clase de las bases que contiene a los metales y los álcalis, agregando la cualidad apariencia con el valor brillante para los metales.

[17] Estrictamente hablando, no todas las clases, esto es, las colecciones de objetos definidas por una propiedad, son conjuntos, véase Amor Montaño [1997] p.5.

2ci. En cada hecho `reacts` se substituya solo una substancia, de otra forma no podrá hacerse una generalización.

2cii. Debe aplicarse a todos los predicados `reacts`, de otra forma significa que no clasifica adecuadamente a las substancias.

Siguiendo el ejemplo con el estado S2 tenemos que:

(`Reacts Inputs` {HCl. NaOH} `Outputs` {NaCl})
(`Reacts Inputs` {HCl. KOH} `Outputs` {KCl})
(`Reacts Inputs` {HNO_3. NaOH} `Outputs` {$NaNO_3$})
(`Reacts Inputs` {HNO_3. KOH} `Outputs` {KNO_3})

se cambian por:

(`Reacts Inputs` {HCl. NaOH} `Outputs` {*tastes-salty*})
(`Reacts Inputs` {HCl. KOH} `Outputs` {*tastes-salty*})
(`Reacts Inputs` {HNO_3. NaOH} `Outputs` {*tastes-salty*})
(`Reacts Inputs` {HNO_3. KOH} `Outputs` {*tastes-salty*})

2d. Si se violó alguna de las condiciones 2(c)i o 2(c)ii entonces se desecha la clase propuesta y se regresa al paso 2 para intentar la siguiente. En caso contrario continuar.

3. Se aplica `Determine-Quantifier` con la clase C encontrada en el paso 2 a los predicados de forma `reacts`

3a. Calcula | `reacts`|, esto es el número de hechos `reacts`

3b. Se calcula el número de hechos `reacts` después de la substitución, esto es | `reacts{[`x`:=`*tastes-salty*`]` | $x \in$ *tastes-salty* `}` |

3c. Se calcula $|Clase|$, el tamaño de la clase C.

3d. Determinar el cuantificador con la siguiente regla

$$cuantificador = \begin{cases} \forall & \text{si } \frac{|reacts|}{|reacts\{[x{:=}Clase] \div x \in Clase\}|} = |Clase| \\ \exists! & \text{si } \frac{|reacts|}{|reacts\{[x{:=}Clase] \div x \in Clase\}|} = 1 \\ \exists & \text{si } 1 < \frac{|reacts|}{|reacts\{[x{:=}Clase] \div x \in Clase\}|} < |Clase| \end{cases}$$

y se cuantifican los hechos `reacts`. Siguiendo el ejemplo tenemos $\frac{4}{4} = 1$ por lo que corresponde $\exists!$

($\exists!$ *tastes-salty* (`Reacts Inputs` {HCl. NaOH} `Outputs` {*tastes-salty*}))
($\exists!$ *tastes-salty* (`Reacts Inputs` {HCl. KOH} `Outputs` {*tastes-salty*}))
($\exists!$ *tastes-salty* (`Reacts Inputs` {HNO_3. NaOH} `Outputs` {*tastes-salty*}))
($\exists!$ *tastes-salty* (`Reacts Inputs` {HNO_3. KOH} `Outputs` {*tastes-salty*}))

3e. Como se cumplió con la restricción 2(c)ii respecto a que la substitución debe aplicarse a todos los hechos tipo `reacts`, podemos cuantificar universalmente la definición por intensión de la clase C. Siguiendo el ejemplo:

tastes-salty = *clase* (*Tastes*, *Salty*)
tastes-salty = {*x* | (Has-Quality Object {*x*} *Tastes* {*Salty*})}

queda como sigue:

tastes-salty = *clase* (*Tastes*, *Salty*)
(∀ *x* ∈ *tastes-salty* (Has-Quality Object {*x*} *Tastes* {*Salty*}))

aunque GLAUBER usa la siguiente forma:

(∀ *tastes-salty* (Has-Quality Object{*tastes-salty*} *Tastes*{*Salty*}))

4. Prueba la terminación

Si pueden formarse más clases y existen substancias en los predicados *reacts*,

entonces se repite el procedimiento desde el paso 2 buscando en las demás clases propuestas,

en otro caso Termina el procedimiento.

2.4 Un ensayo que muestra limitaciones de GLAUBER

Como se mencionó anteriormente, las limitaciones de GLAUBER se deben en gran parte a que dicho sistema forma clases tomando en cuenta una sola cualidad. Con el fin de mostrar esto, se modificó el ejemplo con el que se presentó GLAUBER agregando reacciones con el ácido –débil– acético (CH_3-CO-OH) que reacciona con ácidos –fuertes– o con álcalis para formar "sales", señalando los hechos nuevos con *:

(Reacts Inputs {HCl, NaOH} Outputs {NaCl})
(Reacts Inputs {HCl, KOH} Outputs {KCl})
(Reacts Inputs {HNO_3, NaOH} Outputs {$NaNO_3$})
(Reacts Inputs {HNO_3, KOH} Outputs {KNO_3})
(Reacts Inputs {HCl, CH_3-CO-OH} Outputs {CH_3-CO-Cl}) *
(Reacts Inputs {HNO_3, CH_3-CO-OH} Outputs {CH_3-CO-NO_3}) *
(Reacts Inputs {NaOH, CH_3-CO-OH} Outputs {CH_3-CO-O-Na}) *
(Reacts Inputs {KOH, CH_3-CO-OH} Outputs {CH_3-CO-O-K}) *

También se agregó la cualidad sabor del ácido acético y las nuevas sales:

(Has-Quality Object {NaOH} *Tastes* {*Bitter*})
(Has-Quality Object {KOH} *Tastes* {*Bitter*})
(Has-Quality Object {HCl} *Tastes* {*Sour*})
(Has-Quality Object {HNO_3} *Tastes* {*Sour*})
(Has-Quality Object {CH_3-CO-OH} *Tastes* {*Sour*}) *

(Has-Quality Object {NaCl} *Tastes* {*Salty*})
(Has-Quality Object {$NaNO_3$} *Tastes* {*Salty*})
(Has-Quality Object {KCl} *Tastes* {*Salty*})
(Has-Quality Object {KNO_3} *Tastes* {*Salty*})
(Has-Quality Object {CH_3-CO-NO_3} *Tastes* {*Salty*}) *
(Has-Quality Object {CH_3-CO-Cl} *Tastes* {*Salty*}) *
(Has-Quality Object {CH_3-CO-O-Na} *Tastes* {*Salty*}) *
(Has-Quality Object {CH_3-CO-O-K} *Tastes* {*Salty*}) *

Incluye además una nueva cualidad *Activity* que se refiere a la fuerza con que reacciona:

(Has-Quality Object {NaOH} *Activity* {*Strong*}) *
(Has-Quality Object {KOH} *Activity* {*Strong*}) *
(Has-Quality Object {HCl} *Activity* {*Strong*}) *
(Has-Quality Object {HNO_3} *Activity* {*Strong*}) *
(Has-Quality Object {CH_3-CO-OH} *Activity* {*Weak*}) *
(Has-Quality Object {NaCl} *Activity* {*Neutral*}) *
(Has-Quality Object {$NaNO_3$} *Activity* {*Neutral*}) *
(Has-Quality Object {KCl} *Activity* {*Neutral*}) *
(Has-Quality Object {KNO_3} *Activity* {*Neutral*}) *
(Has-Quality Object {CH_3-CO-Cl} *Activity* {*Neutral*}) *
(Has-Quality Object {CH_3-CO-NO_3} *Activity* {*Neutral*}) *
(Has-Quality Object {CH_3-CO-O-Na} *Activity* {*Neutral*}) *
(Has-Quality Object {CH_3-CO-O-K} *Activity* {*Neutral*}) *

GLAUBER falla debido a que encuentra leyes incompletas porque tienen constantes. En el siguiente ejemplo, aparecen nombres de las substancias HCl,HNO_3, etc., en lugar del nombre de la clase a la que pertenecen:

$\forall$ *Activity*- *Weak* $\exists$ *Tastes*-*Bitter* $\exists!$ *Activity*-*Neutral*
(Reacts Inputs{*Tastes*-*Bitter*, *Activity*-*Weak*} Outputs{*Activity*-*Neutral*})
$\forall$ *Activity*- *Weak* $\exists!$ *Activity*-*Neutral*
(Reacts Inputs{HNO_3, *Activity*-*Weak*} Outputs{*Activity*-*Neutral*})
$\forall$ *Activity*- *Weak* $\exists!$ *Activity*-*Neutral*
(Reacts Inputs{HCl, *Activity*-*Weak*} Outputs{*Activity*-*Neutral*})
$\exists$ *Tastes*-*Bitter* $\exists!$ *Activity*-*Neutral*
(Reacts Inputs{HNO_3, *Tastes*-*Bitter*} Outputs{*Activity*-*Neutral*})
$\exists$ *Tastes*-*Bitter* $\exists!$ *Activity*-*Neutral*
(Reacts Inputs{HCl, *Tastes*-*Bitter*} Outputs{*Activity*-*Neutral*})

Posteriormente describe las clases por extensión del siguiente modo:

Activity- *Neutral*={NaCl, $NaNO_3$, KCl, KNO_3,
CH_3-CO-Cl, CH_3-CO-NO_3,
CH_3-CO-O-Na, CH_3-CO-O-K}
Tastes-*Bitter*={NaOH, KOH}
Activity- *Weak*={CH_3-CO-OH}

Asimismo se describen intensionalmente con:

$\forall$ *Activity-Neutral* (Has-Quality Object{*Activity-Neutral*} *Activity*{*Neutral*})
$\forall$ *Tastes-Bitter* (Has-Quality Object{*Tastes-Bitter*} *Tastes*{*Bitter*})
$\forall$ *Activity-Weak* (Has-Quality Object{*Activity-Weak*} *Activity*{*Weak*})

Por su parte, SURREALIST logra acertar descubriendo las leyes correctas:

$\forall\ c_1 \in C_1\ \forall\ c_2 \in C_2\ \exists!\ c_3 \in C_3$ (Reacts Inputs $\{c_1, c_2\}$ Outputs $\{c_3\}$)
$\forall\ c_4 \in C_4\ \forall\ c_5 \in C_5\ \exists!\ c_3 \in C_3$ (Reacts Inputs $\{c_4, c_5\}$ Outputs $\{c_3\}$)

La descripción extensional de las clases es la siguiente:

C_1 = {HCl, HNO_3}
C_2 = {CH_3-CO-OH}
C_3 = {$NaCl, NaNO_3, KCl, KNO_3, CH_3$-CO-Cl, CH_3-CO-NO_3, CH_3-CO-O-Na, CH_3-CO-O-K}
C_4 = {$NaOH, KOH$}
$C_5 = C_1 \cup C_2$

Lo anterior tiene su correspondencia en la siguiente descripción intensional:

C_1 = {x | (Has-Quality Object{x} *Tastes*{*Sour*}) $\wedge$(Has-Quality Object{x} *Activity*{*Strong*})}
C_2 = {x | (Has-Quality Object{x} *Tastes*{*Sour*}) $\wedge$(Has-Quality Object{x} *Activity*{*Weak*})}
C_3 = {x | (Has-Quality Object{x} *Tastes*{*Salty*}) $\wedge$(Has-Quality Object{x} *Activity*{*Neutral*})}
C_4 = {x | (Has-Quality Object{x} *Tastes*{*Bitter*})}
C_5 = {x | (Has-Quality Object{x} *Tastes*{*Sour*}) $\wedge$[(Has-Quality Object{x} *Activity*{*Strong*}) $\vee$ (Has-Quality Object{x} *Activity*{*Weak*})]}

Con GLAUBER es imposible descubrir estas leyes, formando las clases únicamente con una cualidad. Lo más que puede obtener es:

$\forall$ *Tastes-Sour* $\forall$ *Tastes-Bitter* $\exists!$ *Activity-Neutral*
(Reacts Inputs{*Tastes-Sour*, *Tastes-Bitter*} Outputs{*Activity-Neutral*})
$\forall$ *Activity-Strong* $\forall$ *Activity-Weak* $\exists!$ *Activity-Neutral*
(Reacts Inputs{*Activity-Strong*, *Activity-Weak*} Outputs{*Activity-Neutral*})

donde la única clase descrita es:

$\forall$ *Activity-Strong* (Has-Quality Object{*Activity-Strong*} *Activity*{*Strong*})

Empero, este conjunto de reglas no describe adecuadamente el fenómeno.

Capítulo 3

SURREALIST

SURREALIST es un método heurístico basado en GLAUBER el cual transforma conjuntos de predicados `Reacts` y `Has-Quality` a un conjunto de clases y reglas generales:

El proceso se encamina a descubrir reglas y clases formuladas a partir de ejemplares observados (hechos), cuya intepretación únicamente corresponda con ejemplares observables[1] (predicciones).[2]

El principal tanteo de las heurísticas de GLAUBER y SURREALIST, consiste en substituir a los objetos en los ejemplares de `Reacts` por variables que toman su valor de las clases obtenidas con la información en los ejemplares de `Has-Quality`, hasta obtener reglas generales que idealmente solo podrán generar lo observable, o dicho más apropiadamente, un conjunto de reglas consistentes con el "mundo real" como intepretación.

Dichas clases son de equivalencia[3] debido a que parten al conjunto de substancias en subconjuntos disjuntos, en función del valor de las cualidades comunes a sus miembros. Podemos describirlas mediante fórmulas cuantificadas como sigue:

[1]Por supuesto que los ejemplares observables (predicciones) incluyen a los observados (hechos).

[2]En lógica, a una *interpretación* que es verdadera se llama *modelo*, pero en otros campos se llama modelo al conjunto de reglas y clases.

[3]Dos substancias s_1 y s_2, son equivalentes $s_1 = s_2$, si las cualidades que parten al conjunto en clases disjuntas, tienen los mismos valores para s_1 y s_2.

$$\text{Ácidos} = \{x \cdot \exists \cdot x \in \text{Substancias} \wedge \texttt{Has-Quality}(\text{Object}(x), Tastes(Sour))\}$$
$$\text{Álcalis} = \{x \cdot \exists \cdot x \in \text{Substancias} \wedge \texttt{Has-Quality}(\text{Object}(x), Tastes(Bitter))\}$$
$$\text{Sales} = \{x \cdot \exists \cdot x \in \text{Substancias} \wedge \texttt{Has-Quality}(\text{Object}(x), Tastes(Salty))\}$$

Las reglas son generalizaciones de los predicados `Reacts` cuantificadas sobre las clases. Esto es, las reglas son casos del predicado `Reacts` aplicado a variables cuantificadas sobre las clases descubiertas:

$$\forall a \in \text{Ácidos} \cdot \forall k \in \text{Álkalis} \cdot \exists s \in \text{Sales} \cdot \texttt{Reacts}(\text{Inputs}(\{a, k\}), \text{Outputs}(\{s\}))$$

3.1 Análisis de la generalización

De entrada, se revisarán de manera abstracta los refinamientos que hace GLAUBER. El proceso de generalización parte de los ejemplares observados:

`Reacts`(Inputs {HCl, NaOH}, Outputs {NaCl})
`Reacts`(Inputs {HCl, KOH}, Outputs {KCl})
`Reacts`(Inputs {HNO_3, NaOH}, Outputs {$NaNO_3$})
`Reacts`(Inputs {HNO_3, KOH}, Outputs {KNO_3})

con los que en un primer intento burdo de generalización obtiene las reglas:

`Reacts` (Inputs *reactivos*, Outputs *productos*)
donde *reactivos*, *productos* $\subset$ Substancias y
Substancias ={HCl,HNO_3,NaOH,KOH,NaCl,KCl,$NaNO_3$,KNO_3}

Con un ligero refinamiento, *reactivos* se restringe a un conjunto con dos substancias, y *productos* a un conjunto con una substancia:

`Reacts` (Inputs $\{a, k\}$, Outputs $\{s\}$)
donde $a, k, s \in$ Substancias y
Substancias ={HCl,HNO_3,NaOH,KOH,NaCl,KCl,$NaNO_3$,KNO_3}

Un nuevo refinamiento precisa cuáles subconjuntos son disjuntos, es decir, qué clases de equivalencia proveen valores a las variables a, k y s. Con el resultado que se muestra a continuación, solo puede reconstruirse el conjunto de observaciones, aunque la información descubierta al separar en clases disjuntas será de utiliad para generalizar la ley:

$\texttt{Reacts}(\text{Inputs}(\{a, k\}), \text{Outputs}(\{s\}))$
donde $a \in \{\text{HCl}, HNO_3\}$
$\quad k \in \{\text{NaOH}, \text{KOH}\}$
y $\quad s \in \{\text{NaCl}, \text{KCl}, NaNO_3, KNO_3\}$

Para generalizar, se busca la descripción *intensional* a partir de sus *extensiones* y los predicados `Has-Quality`:

`Has-Quality`(Object(HCl), Tastes(Sour))
`Has-Quality`(Object(HNO_3}), Tastes(Sour))
`Has-Quality`(Object(NaOH}), Tastes(Bitter))
`Has-Quality`(Object(KOH}), Tastes(Bitter))
`Has-Quality`(Object($NaNO_3$}), Tastes(Salty))
`Has-Quality`(Object(NaCl}), Tastes(Salty))
`Has-Quality`(Object($NaNO_3$}), Tastes(Salty))
`Has-Quality`(Object(KNO_3}), Tastes(Salty)) [4]

GLAUBER, que solo considera una cualidad a la vez, descubre que $x = y \equiv$ `Has-Quality`$(Object\{x\}\ Tastes\{c\}) \wedge$ `Has-Quality`$(Object\{y\}\ Tastes\{c\})$,[5] es la relación de equivalencia correspondiente a la partición encontrada. A continuación se muestra la definición *intensional* de cada clase junto con su *extensión*:

$\{\text{HCl, } HNO_3\} = \{x \cdot \ni \cdot x \in \text{Substancias} \wedge$ `Has-Quality`$(\text{Object}(x), \text{Tastes}(Sour))\}$
$\{\text{NaOH, KOH}\} = \{x \cdot \ni \cdot x \in \text{Substancias} \wedge$ `Has-Quality`$(\text{Object}(x), \text{Tastes}(Bitter))\}$
$\{\text{NaCl,KCl,}NaNO_3, KNO_3\} = \{x \cdot \ni \cdot x \in \text{Substancias} \wedge$ `Has-Quality`$(\text{Object}(x), \text{Tastes}(Salty))\}$

Este es un primer paso hacia la *generalización*. Si la regla inferida es correcta, se cumplirá para nuevas substancias con los sabores ácido, amargo y salado. Por último es necesario refinar la regla generalizada mediante la cuantificación[6], obteniendo las siguientes leyes, que si son correctas, se aplicarán a todas las substancias que pertenezcan a las clases correspondientes:

$\forall a \in C_a \cdot \forall k \in C_k \cdot \exists s \in C_s \cdot$ `Reacts`$(\text{Inputs}(\{a, k\}), \text{Outputs}(\{s\}))$
donde $C_a = \{x \cdot \ni \cdot$ `Has-Quality`$(Object\{x\}\ Tastes\{Sour\})\}$
$C_k = \{x \cdot \ni \cdot$ `Has-Quality`$(Object\{x\}\ Tastes\{Bitter\})\}$
y $C_s = \{x \cdot \ni \cdot$ `Has-Quality`$(Object\{x\}\ Tastes\{Salty\})\}$

tal como se vio en el capítulo 2, que describe la heurística con mayor detalle.

3.2 Generalización en SURREALIST

SURREALIST tiene una heurística más fuerte[7] en comparación con GLAUBER debido a que representa los predicados `Reacts` y `Has-Quality` mediante relaciones. Esto le permite encontrar todas las cualidades que distinguen a

[4] Alternativamente, puede definirse como sigue:

{`Has-Quality`(Object(x), Tastes($Sour$)) $\cdot \ni \cdot x \in$ {HCl, HNO_3}}∪
{`Has-Quality`(Object(x), Tastes($Bitter$)) $\cdot \ni \cdot x \in$ {NaOH, KOH}}∪
{`Has-Quality`(Object(x), Tastes($Salty$)) $\cdot \ni \cdot x \in$ {NaCl, KCl, $NaNO_3$, KNO_3}}

[5] $x = y$ corresponde al predicado *"x sabe igual que y"*.

[6] Véase §2.1.2.

[7] En el capítulo 1 se explica en qué sentido se habla de heurísticas *fuertes* o *débiles*.

las substancias en cada ejemplar de `Reacts`. En lugar de buscar la cualidad con mayor puntaje, como GLAUBER, SURREALIST busca al mismo tiempo todas las cualidades que diferencian a las substancias. A pesar de la aparente complejidad, ello es posible gracias al *álgebra relacional*, que cuenta con operaciones entre relaciones. Dichas operaciones permiten la búsqueda de todas las cualidades que distinguen a las substancias en todos los ejemplares del predicado `Reacts`, en un solo paso.[8] Estas son precisamente las cualidades que forman las clases de equivalencia.[9]

3.2.1 Transformación de predicados a relaciones

Es necesario traducir la representación de predicados empleada por GLAUBER[10] a relaciones. Cada paso de la transformación produce información que permite revertir el proceso. Esta información también sirve para analizar las *dependencias funcionales*[11], que a su vez servirán para cuantificar las reglas. Otra información de utilidad es el tamaño de los conjuntos *inputs* y *outputs* en `Reacts`,[12] la cual servirá para encontrar las cualidades que determinan las clases.

La traducción de los predicados `Has-Quality` puede hacerse en un paso,

[8]Sin hacer un análisis profundo de la complejidad, pero conociendo la complejidad de las operaciones del álgebra relacional, puede preverse que sea del orden $O(n \log n)$ o en el peor de los casos $O(n^2)$.

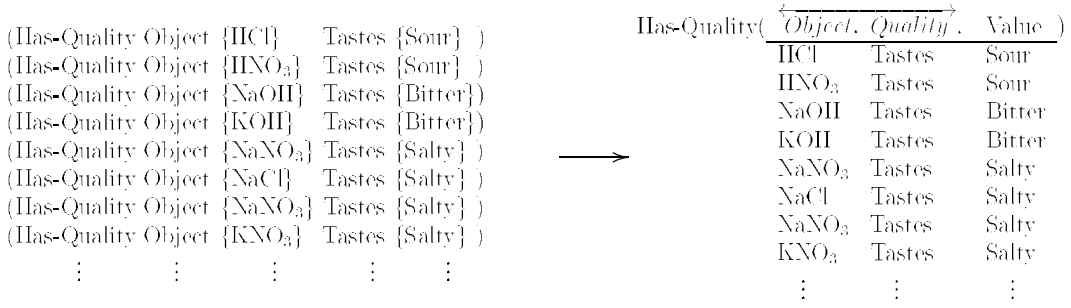
[9]Esto es claramente una ventaja sobre GLAUBER, que representa los predicados mediante listas formando las clases probando una por una las cualidades. Parece que dicha representación se debe a un requerimiento de PRISM, el lenguaje orientado a resolución de problemas en que fue escrito GLAUBER.

[10]Véase §2.1.1.

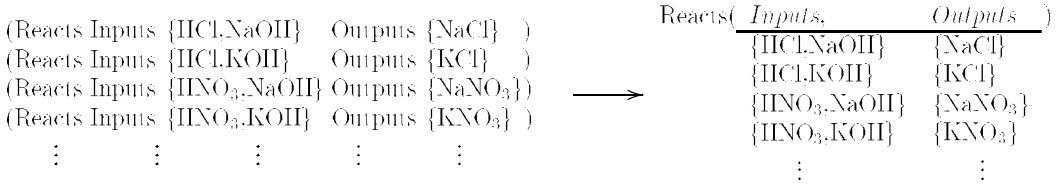
[11]El apéndice B es una breve introducción al *álgebra relacional*, en donde se incluye la notación empleada en esta tesis.

[12]Podría ser distinto en cada *tupla*, lo que permitirá tratar casos más complejos donde no en todos participa el mismo número de substancias en la reacción. Estos se consideran homónimos, ya que el mismo predicado genera relaciones de distinta aridad. Los homónimos pueden tratarse subdividiendo el problema con una estrategia *divide y conquista*, la resolución de cada subproblema generará cúmulos de reglas y clases.

y la *llave*[13] se determina *a priori* por el significado del predicado:



Por su parte, la traducción de los predicados `Reacts` se hace en varias etapas. Primero como un predicado que tiene como entradas dos conjuntos, correspondientes a los reactivos y a los productos[14]:



Colateralmente se analizan las *dependencias funcionales*, para determinar la(s) *llave*(s), obteniéndose alguno de los siguientes esquemas:

$$\texttt{Reacts}(\overleftrightarrow{Inputs, Outputs}),\ \texttt{Reacts}(\overleftrightarrow{Inputs},\ \overleftrightarrow{Outputs}),$$
$$\texttt{Reacts}(\overleftrightarrow{Inputs}, Outputs),\ \text{o}\ \texttt{Reacts}(Inputs,\ \overleftrightarrow{Outputs}).^{15}$$

Todavía es necesario continuar con la transformación, hasta obtener una relación sin conjuntos[16]. Para no perder información y para poder implementar SURREALIST, es necesario agregar una llave para enumerar cada ejemplar,[17]

[13] La *llave* se indicará colocando sobre los atributos que la forman una flecha de dos puntas. Por ejemplo: $\overleftrightarrow{a, b, c}$, véase el glosario en la pág. 95 o el apéndice B para más información.

[14] Más adelante se verá cómo mediante una estratégia *divide y conquista* pueden tratarse predicados `Reacts` con más substancias y más conjuntos de substancias.

[15] Se pospuso hasta §3.3 la manera de determinar las dependencias funcionales para no interrumpir la explicación de la representación.

[16] Históricamente se dice que una relación está en *primera forma normal* cuándo todas sus entradas son atómicas (véase Ullman [1982]).

[17] Además de las llaves encontradas en el paso anterior.

y así "sacar" de los conjuntos a los objetos, como se muestra a continuación:

Reacts($\overleftrightarrow{N}$, *Inputs*, *Outputs*)

$\overleftrightarrow{N}$	*Inputs*	*Outputs*
1	{HCl,NaOH}	{NaCl}
2	{HCl,KOH}	{KCl}
3	{HNO_3,NaOH}	{$NaNO_3$}
4	{HNO_3,KOH}	{KNO_3}
⋮	⋮ ⋮	⋮ ⋮

⟶

Reacts(N, *Set*, *Object*)

N	*Set*	*Object*
1	*Inputs*	HCl
1	*Inputs*	NaOH
1	*Outputs*	NaCl
2	*Inputs*	HCl
2	*Inputs*	KOH
2	*Outputs*	KCl
3	*Inputs*	HNO_3
3	*Inputs*	NaOH
3	*Outputs*	$NaNO_3$
4	*Inputs*	HNO_3
4	*Inputs*	KOH
4	*Outputs*	KNO_3
⋮	⋮ ⋮	⋮ ⋮

3.2.2 Búsqueda de clases

En §2.4 se vio un ejemplo con ácidos fuertes y débiles que GLAUBER no puede procesar correctamente. De esos datos se deben inferir dos reglas para `Reacts`:[18] la primera se refiere a objetos de *ácidos*, *álcalis y sales*, mientras que la segunda a objetos de *ácidos fuertes*, *ácidos débiles y sales*:

$\forall a \in$ Ácidos $\cdot \forall k \in$ Álcalis $\cdot \exists! s \in$ Sales $\cdot$ *Reacts*(*Inputs*$\{a, k\}$, *Outputs*$\{s\}$), y
$\forall a_f \in$ Ácidos-Fuertes $\cdot \forall a_w \in$ Ácidos-Débiles $\cdot \exists! s \in$ Sales $\cdot$ *Reacts*(*Inputs*$\{a_f, a_w\}$, *Outputs*$\{s\}$).

SURREALIST puede encontrarlas porque enfoca su búsqueda en las cualidades que diferencian a las substancias involucradas en cada ejemplar de reacción. Lo hace *juntando*[19] las relaciones `Reacts` y `Has-Quality`, y busca las cualidades que aparecen con distinto valor en todas las substancias de cada ejemplar. La siguiente tabla muestra el *juntado natural* de reacciones y cualidades, en donde se marcó con "◁" a las tuplas con cualidades que diferencian a las substancias de cada ejemplar obteniéndose la relación:[20]

[18] Lo que podría verse como un caso de homonimia. Los partidarios de la *programación orientada a objetos* dirían que está sobrecargada. Este enfoque podría beneficiarse de SURREALIST si se usara para inferir clases. En la *programación orientada a objetos* llaman clases tanto a las estructuras de datos, como a las clases de equivalencia.

[19] Los detalles de las operaciones de juntado natural y otras del *álgebra relacional* se explican en el apéndice B.

[20] En realidad se seleccionan las tuplas que tienen una cuenta igual a la del número de objetos por tupla, como se muestra en la página 85.

N	*Set*	*Object*	*Quality*	*Value*	
1	Inputs	HCl	Tastes	Sour	◁
1	Inputs	HCl	Activity	Strong	
1	Inputs	NaOH	Tastes	Bitter	◁
1	Inputs	NaOH	Activity	Strong	
1	Outputs	NaCl	Tastes	Salty	◁
1	Outputs	NaCl	Activity	Neutral	
2	Inputs	HCl	Tastes	Sour	◁
2	Inputs	HCl	Activity	Strong	
2	Inputs	KOH	Tastes	Bitter	◁
2	Inputs	KOH	Activity	Strong	
2	Outputs	KCl	Tastes	Salty	◁
2	Outputs	KCl	Activity	Neutral	
3	Inputs	HNO_3	Tastes	Sour	◁
3	Inputs	HNO_3	Activity	Strong	
3	Inputs	NaOH	Tastes	Bitter	◁
3	Inputs	NaOH	Activity	Strong	
3	Outputs	$NaNO_3$	Tastes	Salty	◁
3	Outputs	$NaNO_3$	Activity	Neutral	
4	Inputs	HNO_3	Tastes	Sour	◁
4	Inputs	HNO_3	Activity	Strong	
4	Inputs	KOH	Tastes	Bitter	◁
4	Inputs	KOH	Activity	Strong	
4	Outputs	KNO_3	Tastes	Salty	◁
4	Outputs	KNO_3	Activity	Neutral	
5	Inputs	CH_3-CO-O-H	**Activity**	**Weak**	◁
5	Inputs	CH_3-CO-O-H	**Tastes**	**Sour**	◁
5	Inputs	NaOH	**Activity**	**Strong**	◁
5	Inputs	NaOH	**Tastes**	**Bitter**	◁
5	Outputs	CH_3-CO-O-Na	**Activity**	**Neutral**	◁
5	Outputs	CH_3-CO-O-Na	**Tastes**	**Salty**	◁
6	Inputs	CH_3-CO-O-H	Activity	Weak	◁
6	Inputs	CH_3-CO-O-H	Tastes	Sour	◁
6	Inputs	KOH	Activity	Strong	◁
6	Inputs	KOH	Tastes	Bitter	◁
6	Outputs	CH_3-CO-O-K	Activity	Neutral	◁
6	Outputs	CH_3-CO-O-K	Tastes	Salty	◁

7	Inputs	HCl	Tastes	Sour	
7	Inputs	HCl	Activity	Strong	◁
7	Inputs	CH_3-CO-O-H	Tastes	Sour	
7	Inputs	CH_3-CO-O-H	Activity	Weak	◁
7	Outputs	CH_3-CO-Cl	Tastes	Salty	
7	Outputs	CH_3-CO-Cl	Activity	Neutral	◁

8	Inputs	HNO_3	Tastes	Sour	
8	Inputs	HNO_3	Activity	Strong	◁
8	Inputs	CH_3-CO-O-H	Tastes	Sour	
8	Inputs	CH_3-CO-O-H	Activity	Weak	◁
8	Outputs	CH_3-CO-NO_3	Tastes	Salty	
8	Outputs	CH_3-CO-NO_3	Activity	Neutral	◁

Para separar las clases de equivalencia y formular las leyes, se toma un ejemplar como muestra: el que tenga más cualidades es buen candidato, debido a que podría contener las *subclases*[21] de una jerarquía. Se selecciona alguna muestra al azar entre los ejemplares que podrían pertenecer a una subclase. Si se escoge el ejemplar 5 como muestra, se procesa proyectando $\{Object, Quality, Value\}$, sustituyendo a los objetos por nombres[22] para las clases de equivalencia,[23] como sigue:

N Set	*Object*	*Quality*	*Value*		*Class*	*Quality*	*Value*
5 Inputs	CH_3-CO-O-H	**Activity**	**Weak**		Ácido-Débil	**Activity**	**Weak**
5 Inputs	CH_3-CO-O-H	**Tastes**	**Sour**		Ácido-Débil	**Tastes**	**Sour**
5 Inputs	NaOH	**Activity**	**Strong**	⟶	Álcali-Fuerte	**Activity**	**Strong**
5 Inputs	NaOH	**Tastes**	**Bitter**		Álcali-Fuerte	**Tastes**	**Bitter**
5 Outputs	CH_3-CO-O-Na	**Activity**	**Neutral**		Sal-Neutra	**Activity**	**Neutral**
5 Outputs	CH_3-CO-O-Na	**Tastes**	**Salty**		Sal-Neutra	**Tastes**	**Salty**

de donde se obtiene la regla para definir intensionalmente las clases:

Ácido-Débil = $\{x \div$ `Has-Quality`$(Object\{x\}\ Tastes\{Sour\}) \wedge$ `Has-Quality`$(Object\{x\}\ Activity\{Weak\})\}$
Sal-Neutra = $\{x \div$ `Has-Quality`$(Object\{x\}\ Tastes\{Salty\}) \wedge$ `Has-Quality`$(Object\{x\}\ Activity\{Neutral\})\}$
Álcali-Fuerte = $\{x \div$ `Has-Quality`$(Object\{x\}\ Tastes\{Bitter\}) \wedge$ `Has-Quality`$(Object\{x\}\ Activity\{Strong\})\}$

y la relga:

[21] Las *subclases* tienen más cualidades que las superclases; por ejempo la clase de los ácidos tiene como cualidad el sabor, mientras que las subclases: ácido-fuerte y ácido-débil se caracterizan con las cualidades sabor y reactividad.

[22] El nombre puede ser arbitrario, pero es conveniente formarlo con una composición de las cualidades que determinan la clase como se hizo en el ejemplo. Sorpendentemente, en este caso, son muy similares a los empleados por los químicos.

[23] En matemáticas se acostumbra nombrar a una clase de equivalencia escribiendo un elemento de la clase entre corchetes. Las clases de este caso puedrían llamarse {[CH_3-CO-O-H], [NaOH], [CH_3-CO-O-Na]}, siendo [NaOH] = [KOH].

Reacts($Inputs\{a, k\}$ $Outputs\{s\}$)
donde $a \in$ Ácido-Débil, $k \in$ Álkali-Fuerte y $s \in$ Sal-Neutra

Las clases se separan dividiendo la relación entre las clases obtenidas de la muestra. La operación de división $r_1 \div r_2$, donde los atributos del divisor r_2 son un subconjunto del dividendo r_1, produce como resultado un cociente q, que es una relación con los atributos de r_2 sin los atributos de r_1, de tal forma que $q \times r_2 = r_1 \setminus r$ donde r es el residuo. El cociente contiene a todas las tuplas del dividendo r_1 que tienen todos los valores del divisor r_2. Así, el cociente de la división es:

N Set	*Object*	*Class*
5 Inputs	CH_3-CO-O-H	Ácido Débil
5 Inputs	NaOH	Álcali Fuerte
5 Outputs	CH_3-CO-O-Na	Sal Neutra
6 Inputs	CH_3-CO-O-H	Ácido Débil
6 Inputs	KOH	Álcali Fuerte
6 Outputs	CH_3-CO-O-K	Sal Neutra

subconjunto del que se obtiene la regla:

Reacts($Inputs\{a, k\}$ $Outputs\{s\}$)
donde $a \in \{CH_3\text{-CO-O-H}\}$, $k \in \{NaOH, NaOH\}$ y $s \in \{CH_3\text{-CO-O-Na}, CH_3\text{-CO-O-K}\}$

También se obtiene el residuo:[24]

N	*Set*	*Object*	*Quality*	*Value*	
1	Inputs	HCl	Tastes	Sour	◁
1	Inputs	NaOH	Tastes	Bitter	◁
1	Outputs	NaCl	Tastes	Salty	◁
2	Inputs	HCl	Tastes	Sour	◁
2	Inputs	KOH	Tastes	Bitter	◁
2	Outputs	KCl	Tastes	Salty	◁
3	Inputs	HNO_3	Tastes	Sour	◁
3	Inputs	NaOH	Tastes	Bitter	◁
3	Outputs	$NaNO_3$	Tastes	Salty	◁
4	Inputs	HNO_3	Tastes	Sour	◁
4	Inputs	KOH	Tastes	Bitter	◁
4	Outputs	KNO_3	Tastes	Salty	◁

[24] Observe que puede revertirse el proceso, porque se cumple la propiedad $r_1 = (q \times r_2) \cup r$.

7	Inputs	HCl	Activity	Strong	◁
7	Inputs	CH_3-CO-O-H	Activity	Weak	◁
7	Outputs	CH_3-CO-Cl	Activity	Neutral	◁
8	Inputs	HNO_3	Activity	Strong	◁
8	Inputs	CH_3-CO-O-H	Activity	Weak	◁
8	Outputs	CH_3-CO-NO_3	Activity	Neutral	◁

Como el residuo no es la relación vacía $\varnothing$, se vuelve a seleccionar una muestra repitiendo el procedimiento hasta obtener la relación vacía $\varnothing$, como residuo. En el estado actual del cómputo se obtienen las siguientes relaciones:

N	*Set*	*Object*	*Class*
1	Inputs	HCl	Ácido-Fuerte
1	Inputs	NaOH	Álcali-Fuerte
1	Outputs	NaCl	Sal-Neutra
2	Inputs	HCl	Ácido-Fuerte
2	Inputs	KOH	Álcali-Fuerte
2	Outputs	KCl	Sal-Neutra
3	Inputs	HNO_3	Ácido-Fuerte
3	Inputs	NaOH	Álcali-Fuerte
3	Outputs	$NaNO_3$	Sal-Neutra
4	Inputs	HNO_3	Ácido-Fuerte
4	Inputs	KOH	Álcali-Fuerte
4	Outputs	KNO_3	Sal-Neutra

N	*Set*	*Object*	*Class*
5	Inputs	CH_3-CO-O-H	Ácido-Débil
5	Inputs	NaOH	Álcali-Fuerte
5	Outputs	CH_3-CO-O-Na	Sal-Neutra
6	Inputs	CH_3-CO-O-H	Ácido-Debil
6	Inputs	KOH	Álcali-Fuerte
6	Outputs	CH_3-CO-O-K	Sal-Neutra

N	*Set*	*Object*	*Class*
7	Inputs	HCl	Ácido-Fuerte
7	Inputs	CH_3-CO-O-H	Ácido-Débil
7	Outputs	CH_3-CO-Cl	Sal-Neutra
8	Inputs	HNO_3	Ácido-Fuerte
8	Inputs	CH_3-CO-O-H	Ácido-Débil
8	Outputs	CH_3-CO-NO_3	Sal-Neutra

De cada cociente y cada divisor se obtienen una regla y un conjunto de clases. Los tres que se obtienen del ejemplo anterior pueden fundirse en las siguientes reglas encontradas por SURREALIST:

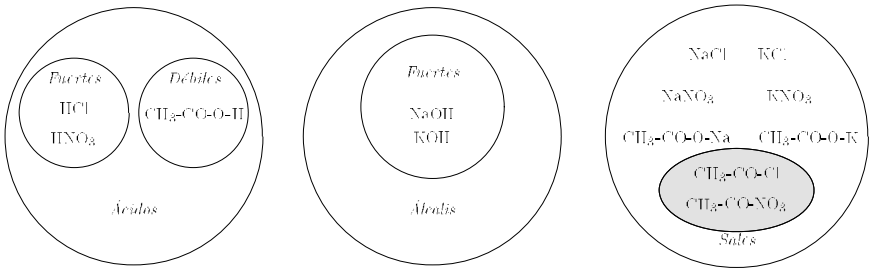
$\texttt{Reacts}(Inputs\{a,k\}, Outputs\{s\})$ y $\texttt{Reacts}(Inputs\{a_s,a_w\}, Outputs\{s\})$
donde $a \in Acids$, $k \in Alkalis$, $s \in Salts$, $a_s \in Strong.Acids$ and $a_w \in Weak.Acids$, además $Acids = Strong.Acids \cup Weak.Acids$ y $Strong.Acids \cap Weak.Acids = \varnothing$.

Las clases descritas intensionalmente son:

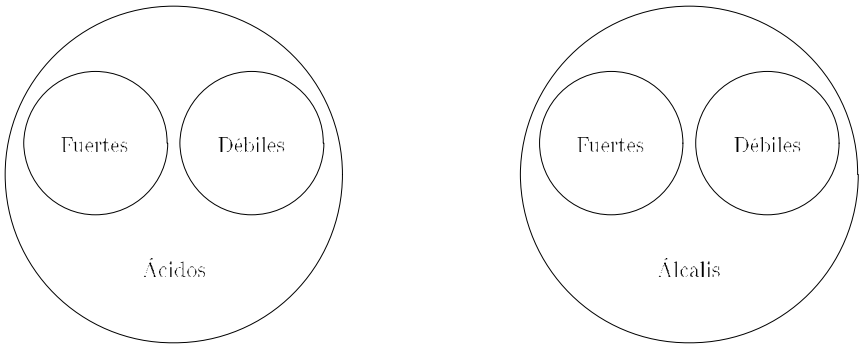
$Acids = \{x \cdot \ni \cdot \texttt{Has-Quality}(Object\{x\}\ Tastes\{Sour\})\}$
$Alkalis = \{x \cdot \ni \cdot \texttt{Has-Quality}(Object\{x\}\ Tastes\{Bitter\})\}$
$Salts = \{x \cdot \ni \cdot \texttt{Has-Quality}(Object\{x\}\ Tastes\{Salty\})\}$
$Strong.Acids = \{x \cdot \ni \cdot x \in Acids \wedge \texttt{Has-Quality}(Object\{x\}\ Activity\{Strong\})\}$
$Weak.Acids = \{x \cdot \ni \cdot x \in Acids \wedge \texttt{Has-Quality}(Object\{x\}\ Activity\{Weak\})\}$

y extensionalmente:

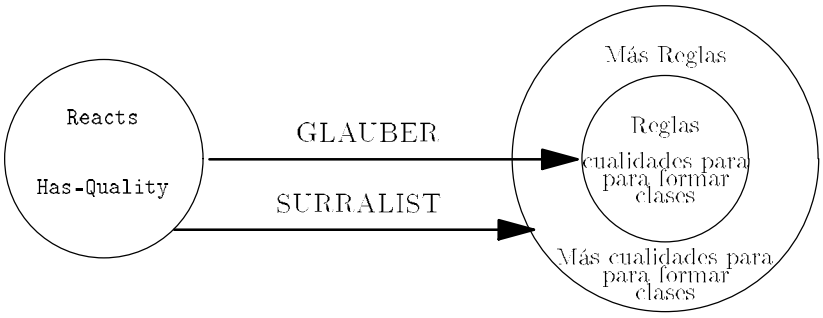
$Strong\_Acids = \{\mathrm{HCl}, \mathrm{HNO_3}\}$
$Weak\_Acids = \{\mathrm{CH_3\text{-}CO\text{-}O\text{-}H}\}$
$Acids = Strong\_Acids \cup Weak\_Acids$
$Alkalis = \{\mathrm{NaOH}, \mathrm{KOH}\}$
$Salts = \{\mathrm{NaCl}, \mathrm{KCl}, \mathrm{NaNO_3}, \mathrm{KNO_3}, \mathrm{CH_3CO\text{-}O\text{-}Na}, \mathrm{CH_3CO\text{-}O\text{-}K}, \mathrm{CH_3CO\text{-}Cl}, \mathrm{CH_3CO\text{-}NO_3}\}$



GLAUBER fracasa, porque solo puede encontrar una clase que comprende a todos los ácidos, SURREALIST tiene éxito al distinguir *ácidos fuertes* y *débiles*, descubriendo que hay reacciones entre dichos ácidos, que son subclases de los *ácidos* (el producto de dichas reacciones se muestra sombreado en la figura anterior). Si los datos incluyeran ejemplares de `Reacts` y `Has-Quality` con *álcalis fuertes* y *débiles*, también los descubriría:



El ejemplo anterior muestra que, aunque el dominio de SURREALIST sea igual al de GLAUBER, el codomino no es el mismo. Debido a que el resultado de SURREALIST es más refinado, encuentra distintas leyes `Reacts` para objetos de distintas clases. Además, las clases descubiertas por SURREALIST conforman varios cúmulos que pueden guardar una relación jerárquica; en cambio GLAUBER, si llega a tener éxito, sólo descubrirá un cúmulo. El codominio de GLAUBER es un subconjunto del codominio de SURREALIST, como muestra el siguiente diagrama que representa a ambas funciones:



La representación empleada por SURREALIST le permite procesar un conjunto de entrada más general que el de GLAUBER. El dominio de SURREALIST es un superconjunto del dominio de GLAUBER, porque puede incluir y analizar más predicados del tipo `Reacts` con un número arbitrario de objetos en los conjuntos y con un número arbitrario de conjuntos relacionados,[25] incluso con nombres distintos a `Reacts`. El siguiente diagrama muestra cómo predicados no comprendidos en el dominio de GLAUBER, pueden transformarse en elementos de la relación con que se representa la información en SURREALIST:

R_1(A {a_1.a_2}. B {b_1}).
R_2(A {a_1.a_3.a_4}. B {b_2.b_3}).
R_3(A {a_1}.B {b_1.b_2}.C {c_1.c_2.c_3})

θ_1 →

RFacts(N, *Predicate*, *Set*, *Object*)

N	*Predicate*	*Set*	*Object*
1	R_1	A	a_1
1	R_1	A	a_2
1	R_1	B	b_1
2	R_2	A	a_1
2	R_2	A	a_3
2	R_2	A	a_4
2	R_2	B	b_2
2	R_2	B	b_3
3	R_3	A	a_1
3	R_3	B	b_1
3	R_3	B	b_2
3	R_3	C	c_1
3	R_3	C	c_2
3	R_3	C	c_3
⋮	⋮	⋮	⋮

θ_2 ↓

θ_3 ↗

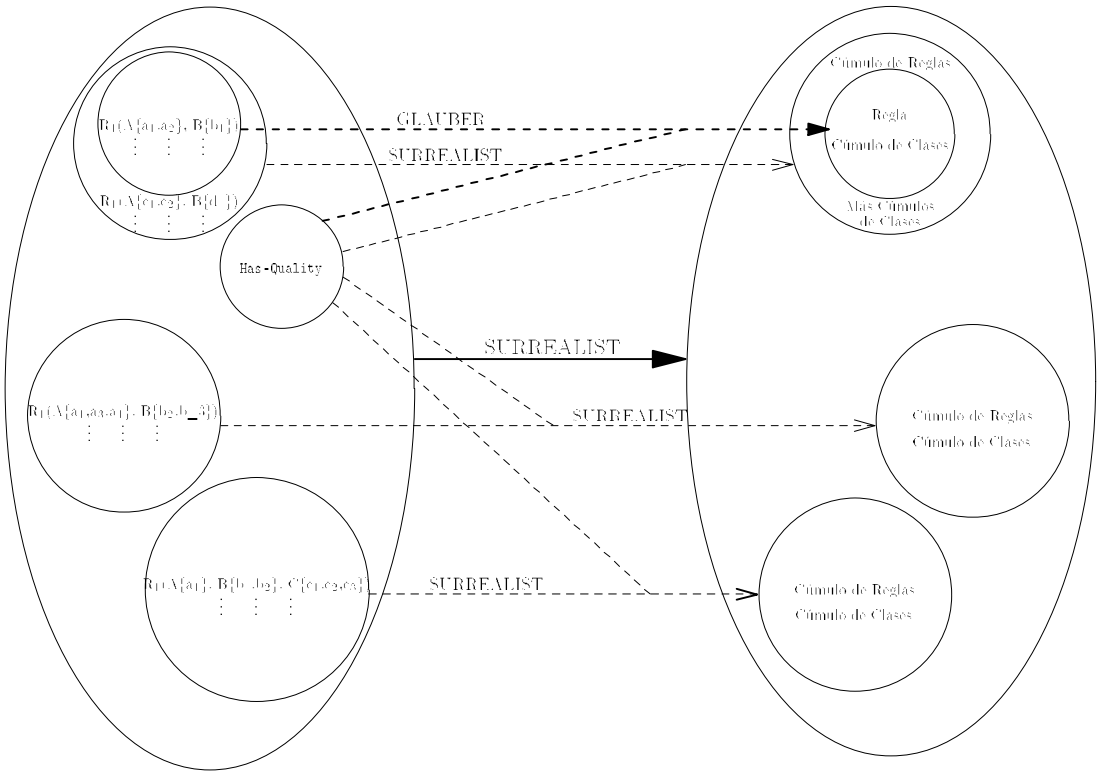
$R_1(A, B) = \{(\{a_1, a_2\}, \{b_1\})\}$
$R_2(A, B) = \{(\{a_1, a_3, a_4\}, \{b_2, b_3\})\}$
$R_3(A, B, C) = \{(\{a_1\}, \{b_1, b_2\}, \{c_1, c_2, c_3\})\}$

Hay que observar que se agregó un atributo para el nombre del predicado, lo que permite representar predicados del tipo `Reacts` con distintos nombres. Otra ventaja es que vincula un mapeo uno-a-uno entre las distintas representaciones, abriéndose la posibilidad para transformar una base de datos relacional para su análisis. La tranformación θ_1 puede hacerse en dos pasos. Aunque se cumple la propiedad $\theta_1 = \theta_2 \circ \theta_3$, es conveniente aprovechar la información producida por θ_2 para determinar más adelante los

[25] También puede ser solo un conjunto.

cuantificadores, como las *llaves* `Reacts`($\overleftrightarrow{Inputs}$, $\overleftrightarrow{Outputs}$) y las *dependencias funcionales* $\{Inputs \rightarrow Outputs, Outputs \rightarrow Inputs\}$ en el ejemplo anterior.[26]

El proceso es del estilo *divide y conquista*: se aplica el método SURREALIST a cada variante de predicados tipo `Reacts`.[27] En el siguiente diagrama se ve que el dominio de SURREALIST es un superconjunto del de GLAUBER, quedando este como uno de los casos en que se divide la entrada:



3.2.3 Posibles resultados colaterales de SURREALIST

El procedimiento anterior podría extenderse para descubrir más leyes. Aunque por el momento no se dispone de dicho método, es posible obtener información que un investigador puede analizar. Un caso son algunas cualidades que SURREALIST desecha al buscar clases, porque son insuficientes para dis-

[26] He analizado la posibilidad de implementar todo el método en un solo procedimiento, y parece viable. Una forma es plantear el problema acudiendo a mi experiencia en la programación imperativa, lo que permite esbozar un programa muy eficiente de complejidad $O(n \log m)$ donde n es el número de ejemplares reacts y m el número de clases. El otro método es transformarlo con reglas que preseven la semántica. La primera aproximación tiene el peligro de no respetar la especificación, la segunda es más correcta, pero más laboriosa, lo que puede generar errores si se realiza manualmente.

[27] Puede aplicarse de manera simultanea.

criminar distintas clases de substancias, pero que son útiles para realizar experimentos que permitan afinar la teoría.

Entre la información desechada hay cualidades que son propias de los conjuntos relacionados, es decir que aparecen con el mismo valor para todas las substancias de un conjunto dado pero con distinto valor en los demás conjuntos. Si vemos a los conjuntos como distintos estados, por ejemplo antes y después de la reacción, pueden interpretarse como cualidades que caracterizan a dichos estados resaltando características que permiten plantear nuevas hipótesis al investigador.

Dichas cualidades que llamaremos "corazonadas" se obtienen seleccionando las tuplas que tengan una cuenta igual al número de objetos en el conjunto. En el ejemplo marcaremos con "♡" las del conjunto *Inputs* y con "♡̸" a las del conjunto *Outputs*[28] con lo que obtenienen las siguientes "corazonadas":

N	*Set*	*Object*	*Quality*	*Value*	
1	Inputs	HCl	Activity	Strong	♡
1	Inputs	NaOH	Activity	Strong	♡
1	Outputs	NaCl	Activity	Neutral	♡̸
2	Inputs	HCl	Activity	Strong	♡
2	Inputs	KOH	Activity	Strong	♡
2	Outputs	KCl	Activity	Neutral	♡̸
3	Inputs	HNO_3	Activity	Strong	♡
3	Inputs	NaOH	Activity	Strong	♡
3	Outputs	$NaNO_3$	Activity	Neutral	♡̸
4	Inputs	HNO_3	Activity	Strong	♡
4	Inputs	KOH	Activity	Strong	♡
4	Outputs	KNO_3	Activity	Neutral	♡̸
7	Inputs	HCl	Tastes	Sour	♡
7	Inputs	CH_3-CO-O-H	Tastes	Sour	♡
7	Outputs	CH_3-CO-Cl	Tastes	Salty	♡̸
8	Inputs	HNO_3	Tastes	Sour	♡
8	Inputs	CH_3-CO-O-H	Tastes	Sour	♡
8	Outputs	CH_3-CO-NO_3	Tastes	Salty	♡̸

[28]Puede usarse unicamente "♡" sin que exista confusión dado que el atributo *Set* los desambigua. En un caso con más conjuntos sería mejor optar por un solo símbolo.

Al igual que las tuplas que encuentran las clases de ácidos, álcalis y sales, puede seleccionarse un ejemplar y aplicarle la división de relaciones así los ejemplares 1-4 producen la observación de que una mezcla de substancias fuertemente activas tiende a transformarse en una mezcla de sustancias neutras. Al revisar los resultados, puede verse que es congruente con la regla: los ácidos reaccionan con álcalis formando sales. Las "corazonadas" de las tuplas 7-8 permiten ver que los ácidos fuertes reaccionan con los ácidos débies formando sales, pero su análisis llevan a la pregunta: "Asi como existen ácidos fuertes y ácidos débiles, ¿Hay álcalis fuertes y álcalis débiles?". Con más información, por ejemplo la temperatura también podría observarse que hay reacciones *endotérmicas* y *exotérmicas*.

Por otra parte, aunque existe una relación parcial de orden en las clases descubiertas y que en el ejemplo visto no representa ningún problema ordenarlas. Es posible que las "corazonadas" permitan aprovechar información para reducir la complejidad del ordenamiento.

3.3 Cuantificación en SURREALIST

El primer paso hacia la cuantificación de los predicados consiste en analizar las dependencias entre conjuntos de atributos en el esquema de relación correspondiente.

Para la relación $\texttt{Has-Quality}(Object, Quality, Value)$, dicha dependencia es evidente, ya que por definición cada objeto tiene para cada cualidad un valor único. La relación puede verse como la función:

$$\texttt{Has-Quality} : Object \times Quality \rightarrow Value$$

En la terminología de la teoría de bases de datos relacionales se dice que los atributos $Object$ y $Quality$ *determinan funcionalmente* al atributo $Value$, o que el atributo $Value$ *depende funcionalmente* de los atributos $Object$ y $Quality$, mediante la notación $\{Object, Quality\} \rightarrow \{Value\}$.

El dominio de la función corresponde a la proyección de los atributos $Object$ y $Quality$, se escribe $\pi_{Object,Quality}(\texttt{Has-Quality})$. Cualquier conjunto de atributos que genere mediante la proyección al dominio de una función en una relación, se denomina *llave*. Una *llave* se califica como *primaria* cuando no es superconjunto propio de otra y se indica colocando una flecha de doble punta sobre los atributos que la forman. Con lo que el esquema correcto es $\texttt{Has-Quality}(\overleftrightarrow{Object, Quality}, Value)$.

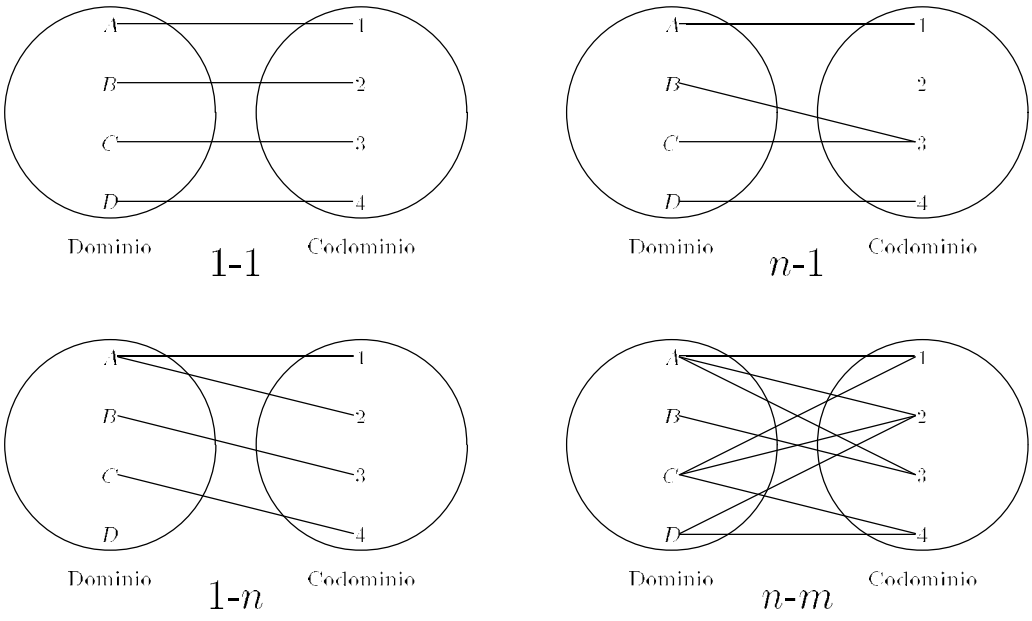
La propiedad de que el dominio de una función[29] tenga la misma cardinalidad que la función, $|\pi_{Object,Quality}(\texttt{Has-Quality})| = |\texttt{Has-Quality}|$, permite

[29]Debido a que se asume un mundo cerrado, las funciones son totales. No existen tuplas con atributos indefinidos, como $(llave, \perp)$.

encontrar las dependencias funcionales, y en consecuencia las llaves primarias.[30] La siguiente función hace posible localizar las llaves y compararlas con los atributos pertenecientes al codominio. Como el codominio corresponde con el rango, llamaremos $dom(R)$ y $ran(R)$ a las proyecciones del dominio y del rango:

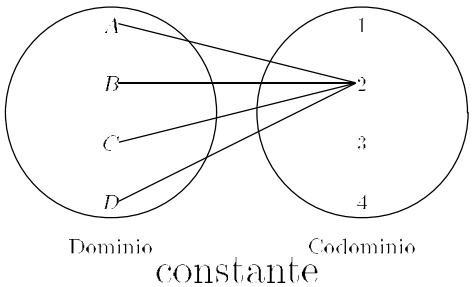
$$keys(R) = \begin{cases} R(\overleftrightarrow{d}, \overleftrightarrow{c}), & \text{si } |dom(R)| = |R| = |ran(R)| \\ R(\overleftrightarrow{d}, c), & \text{si } |dom(R)| = |R| > |ran(R)| \\ R(d, \overleftrightarrow{c}), & \text{si } |dom(R)| < |R| = |ran(R)| \\ R(\overleftrightarrow{d, c}), & \text{si } |dom(R)| < |R| \wedge |ran(R)| < |R| \end{cases}$$

donde $dom(R) = \{x \mid \exists\, (x, y) \in R\}$ y $ran(R) = \{y \mid \exists\, (x, y) \in R\}$, en términos del álgebra relacional $dom(R) = \pi_1(R)$ y $ran(R) = \pi_2(R)$, lo que nos servirá para determinar los cuantificadores. A continuación se muestran las distintas relaciones de cardinalidades entre conjuntos:

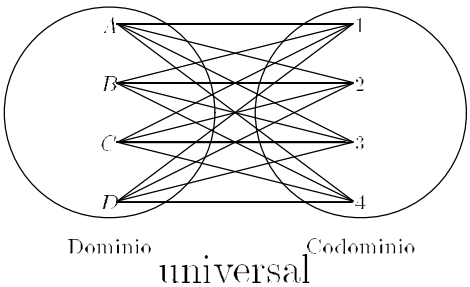


donde las relaciones corresponden a los esquemas 1-1 con $R(\overleftrightarrow{d}, \overleftrightarrow{c})$, n-1 con $R(\overleftrightarrow{d}, c)$, 1-$n$ con $R(d, \overleftrightarrow{c})$, y n-m con $R(\overleftrightarrow{d, c})$. Hay dos casos especiales, la función constante y la relación universal:

[30] Calcular la cardinalidad de todas las proyecciones tiene un costo computacional de $O(2^n)$, donde n es el número de atributos. Pero el panorama no es tan pesimista como parece, hay heurísticas más eficientes.



constante



universal

que corresponden a los esquemas $R(\overleftrightarrow{d,\ c})$ y $R(\overleftrightarrow{d,\ c})$, respectivamente, solo que tienen como característica que $|dom(R)| = |R| \geq |ran(R)| = 1$ y $|R| = |dom(R)| \times |ran(R)|$. Una vez que se conoce el tamaño de las relaciones y sus proyecciones, pueden determinarse las dependencias funcionales y con estas la cuantificación. Aplicando la regla:

$$quant(R(d,c)) = \begin{cases} \forall x\, \exists! y\, R(x,y) \wedge \forall y\, \exists! x\, R(x,y), & \text{si } |dom(R)| = |R| = |ran(R)|, \quad (1\text{–}1) \\ \forall x\, \exists! y\, R(x,y), & \text{si } |dom(R)| = |R| > |ran(R)| > 1, \quad (n\text{–}1) \\ \exists! y\, \forall x\, R(x,y). & \text{si } |dom(R)| = |R| > |ran(R)| = 1, \quad \text{(constante)} \\ \forall y\, \exists! x\, R(x,y). & \text{si } 1 < |dom(R)| < |R| = |ran(R)|, \quad (1\text{–}n) \\ \exists! x\, \forall y\, R(x,y), & \text{si } 1 = |dom(R)| < |R| = |ran(R)|, \quad \text{(constante)} \\ \forall x\, \forall y\, R(x,y). & \text{si } |R| = |dom(R)| \times |ran(R)|. \quad \text{(universal)} \\ \exists x\, \exists y\, R(x,y). & \text{si } |R| \leq |dom(R)| \times |ran(R)|. \quad (m\text{–}n) \end{cases}$$

Así, la cuantificación que corresponde a la relación:

$$\texttt{Has-Quality}(\overleftrightarrow{Object,\ Quality,\ Value})$$

Conforme a la reglas anteriores es:

$$\forall\, (Object, Quality) \cdot \exists!\, Value \cdot \texttt{Has-Quality}(Object, Quality, Value)$$

donde $(Object, Quality)$ es una relación, por lo que la cuantificación no es de primer orden. Sin embargo, dado que no hay una dependencia funcional entre ($Object$ y $Quality$), es posible expresarla en lógica de primer orden como sigue:

$$\forall\, Object \cdot \forall\, Quality \cdot \exists!\, Value \cdot \texttt{Has-Quality}(Object, Quality, Value).$$

Como ya se mencionó, una *llave* es *primaria* si no es superconjunto propio de otra *llave*. En el caso anterior, solo se genera una *llave primaria*, pero en el caso de los predicados `Reacts` se generan *llaves primarias* cuyos elementos son conjuntos. Al transformar a relaciones entre objetos pueden encontrarse otras *llaves primarias* contenidas en las anteriores. La siguiente tabla muestra algunas equivalencias para poder cuantificar solo variables de objetos:

$$\begin{array}{llll}
R(\overleftrightarrow{f(\overleftrightarrow{x,y}),z}) & \equiv R(f(\overleftrightarrow{x,y}),z) & \equiv R(\overleftrightarrow{x,y,z}) & \rightarrow \forall x \cdot \forall y \cdot \exists! z \cdot R(x,y,z) \\
R(\overleftrightarrow{f(\overleftrightarrow{x},\overleftrightarrow{y}),z}) & \equiv R(f(\overleftrightarrow{x},\overleftrightarrow{y}),z) & \equiv R(\overleftrightarrow{x},\overleftrightarrow{y},z) & \rightarrow \forall x \cdot \exists! y \cdot \exists! z \cdot R(x,y,z) \wedge \forall y \cdot \exists! x \cdot \exists! z \cdot R(x,y,z) \\
R(\overleftrightarrow{f(\overleftrightarrow{x},y),z}) & \equiv R(f(\overleftrightarrow{x},y),z) & \equiv R(\overleftrightarrow{x},y,z) & \rightarrow \forall x \cdot \exists! y \cdot \exists! z \cdot R(x,y,z) \\
R(\overleftrightarrow{f(x,\overleftrightarrow{y}),z}) & \equiv R(f(x,\overleftrightarrow{y}),z) & \equiv R(x,\overleftrightarrow{y},z) & \rightarrow \forall y \cdot \exists! x \cdot \exists! z \cdot R(x,y,z)
\end{array}$$

Al analizar los predicados `Reacts` con los datos del ejemplo visto en el capítulo 2 se descubre el siguiente esquema:

$$\texttt{Reacts}(\overleftrightarrow{Inputs}, \overleftrightarrow{Outputs})$$

que debería cuantificarse como sigue:

$$\forall Inputs \cdot \exists! Outputs \cdot \texttt{Reacts}(Inputs.Outputs) \wedge \forall Outputs \cdot \exists! Inputs \cdot \texttt{Reacts}(Inputs.Outputs).$$

Al desanidar los conjuntos, se obtiene:

$$\texttt{Reacts}(\overleftrightarrow{Inputs(\text{Ácido}, \text{Álcali})}, \overleftrightarrow{Outputs(\text{Sal})})$$

Al analizar las dependencias entre los objetos de $Inputs$, se descubre el esquema:

$$\texttt{Reacts}(\overleftrightarrow{Inputs(\overleftrightarrow{\text{Ácido}, \text{Álcali}})}, \overleftrightarrow{Outputs(\overleftrightarrow{\text{Sal}})})$$

La cuantificación de este caso puede expresarse en lógica de primer orden, debido a que las dependencias funcionales de $Inputs\{\overleftrightarrow{a, k}\}$ son equivalentes a las llaves $Inputs\{\overleftrightarrow{a, k}\}$, correspondiente a la cuantificación de primer orden:

$$\forall a \in Acids \cdot \forall k \in Alkalis \cdot \exists! s \in Salts \cdot \texttt{Reacts}(Inputs\{a, k\}, Outputs\{s\}) \wedge$$
$$\forall s \in Salts \cdot \exists! a \in Acids \cdot \exists! k \in Alcalis \cdot \texttt{Reacts}(Inputs\{a, k\}, Outputs\{s\})$$

En el caso de los ácidos débiles y fuertes, las dependencias funcionales son las mismas, por lo que se obtiene:

$$\forall a_s \in Strong.Acids \cdot \forall a_w \in Weak.Acids \cdot \exists! s \in Salts \cdot \texttt{Reacts}(Inputs\{a_s, a_w\}.Outputs\{s\}) \wedge$$
$$\forall s \in Salts \cdot \exists! a_s \in Strong.Acids \cdot \exists! a_w \in Weak.Acids \cdot \texttt{Reacts}(Inputs\{a_s, a_w\}.Outputs\{s\})$$

La relación entre dependencias y cuantificación involucra más casos. Entre estas, las multivaluadas y las de juntado. Las mulitvaluadas introducen el cuantificador $\exists$,[31] el predicado R con la cuantificación $\forall x \cdot \exists y \cdot R(x, y)$, corresponde a la dependencia multivaluada $\{x\} \twoheadrightarrow \{y\}$.

El tema no es tan sencillo como parece, en la teoría de bases de datos relacionales se pone mucho cuidado en todas las dependencias. El esquema de la base de datos se transforma reemplazando algunas relaciones por proyecciones de las relaciones, de tal forma que puedan juntarse sin que se pierda información o que se genere información espurea. Dicho procedimiento se llama *normalización*. En un trabajo futuro, SURREALIST podría aplicarse en la inducción del esquema conceptual de la base de datos.

[31] Se dice que hay una dependencia multivaluada $X \twoheadrightarrow Y$, en una relación $r = \{t_1, t_2, t_3, t_4\}$, con esquema R si $X \cap Y = \varnothing$, $Z = R \setminus (X \cup Y)$ si para cada dos tuplas t_1, t_2, en que $t_1.X = t_2.X$, entonces $t_3 = (t_1.X, t_1.Y, t_2.Z)$ y $t_4 = (t_1.X, t_2.Y, t_1.Z)$. Dicho de otra forma: $r = \pi_{X \cup Y}(r) \bowtie \pi_{X \cup Z}(r)$ donde $X \cup Y = \varnothing$ y $Z = R \setminus (X \cup Y)$ sii $X \twoheadrightarrow Y$.

3.4 Resumen de SURREALIST

El método heurístico SURREALIST consiste en los siguientes pasos:[32]

S1. $[Reglas(N, Predicate, Set, Object, Class) := \varnothing]$ Crear la relación que guardará las reglas sin cuantificar.

S2. [`Has-Quality`$(\overleftrightarrow{Object, Quality}, Value)$:= `parseHasQuality`$(Facts)$] Representar los predicados `Has-Quality` mediante la relación con el esquema `Has-Quality`$(\overleftarrow{Object, Quality}, Value)$.

S3. [`Reacts`$(\overleftrightarrow{N, Predicate, Set, Object})$:= `parseReacts`$(Facts)$] Representar los predicados `Reacts` o de tipo `Reacts` mediatne la relación con el esquema `Reacts`$(\overleftarrow{N, Predicate, Set, Object})$.

S4. $[FReacts^* :=$ `Reacts` $\bowtie$ `Has-Quality`$]$ Agregar a las substancias en la relación `Reacts` todas sus cualidades, mediante el (equi)juntado de `Reacts` con `Has-Quality`, obteniendo $FReacts^*(\overleftarrow{N, Predicate, Set, Object}, Quality)$.

S5. $[FReacts := f_e(FReacts^*)]$ Separar las tuplas que tengan cualidades discriminantes de clases.[33]

S6. $[Divisor := \pi_{Class,Quality,Value}(\pi_{Object,Quality,Value}(f_{max}(FReacts)) \bowtie Nombres(\overleftarrow{Object}, Class))]$ Encontrar una relación $Divisor$, para dividir la relación $FReacts$.

6-i. $[E := f_{max}(FReacts)]$ Buscar un ejemplar en $FReacts$, con el mayor número de cualidades discriminantes.

6-ii. $[M := \pi_{Object,Quality,Value}(E)]$ Proyectar los atributos $Object$, $Quality$, $Value$ del ejemplar obtenido en el paso 6i.

[32]Hasta el momento solo se ha aplicado de manera asistida, es decir, de manera interactiva tomando algunas decisiones el usuario de dicho sistema. Lo que no implica que no pueda escribirse un programa que automatice toda la toma de decisiones. Se ha hecho así por falta de tiempo para aprender a programar la interfaz con la base de datos, la creación de nuevos tipos de datos y la implementación de un algorítmo más eficiente para buscar llaves, que podría aprovechar información estadística del manejador de bases de datos, lo que podría limitar su portabilidad.

[33]En la explicación en §3.2.2 se optó por etiquetar las tuplas para facilitar la lectura. Pero también es conveniente separarlas con una función f_e que selecciona las tuplas de $FReacts$ que tengan igual número de objetos por N, $Predicate$, Set que el número de valores por N, $Predicate$, Set, $Object$, $Quality$. En §3.2.2 también se habló de etiquetar las cualidades que podrían discriminar las superclases.

Aquí se omite porque todavía no se tiene un método para manejarlo, pero es viable utilizar algún algoritmo de ordenamiento topológico para buscar una jerarquía de clases, si es que existe.

6-iii. $[Divisor := \pi_{Class,Quality,Value}(M \bowtie Nombres(\overleftrightarrow{Object}, Class))]$ Reemplazar a cada substancia en la relación obtenida en el paso 6ii por un nombre de clase nuevo. Llamaremos a la relación resultante $Divisor$.

S7. $[Q, R := FReacts \div Divisor, FReacts \,\%\, Divisor]$ Calcular el cociente y el residuo de $FReacts$ y $Divisor$.[34]

S8. $[Reglas, Clases := Reglas \cup Q, Clases \cup Divisor]$ Agregar a la relación $Reglas$ el cociente obtenido en el paso 7.

S9. $[FReacts := R]$

S10. [¿$R \neq \varnothing$?] Si R no es el conjunto vacío, regresar al paso 6; si es el conjunto vacío continuar.

S11. $[EClases := \pi_{Object,Class}]$ Separar la extensión de las clases.

S12. $[GReglas := \pi_{Predicate,Set,Class}]$ Separar las reglas generadas.

S13. $[KReglas := K(Reglas)]$ Encuontrar las llaves para poder encontrar los cuantificadores.[35]

S14. $[CReglas := Cuantifica(KReglas)]$ Cuantificar las reglas con las llaves encontradas en $KReglas$ y las equivalencias definidas en la regla *cuant* de la página 49.

S15. [Terminar]

[34] El cálculo simultáneo es más eficiente.

[35] Por el momento no se ha implementado un algoritmo eficiente, por lo que se omite el detalle del procedimiento. El método ineficiente con complejidad $O(2^n)$ donde n es el número de objetos en la relación, consiste en contar todos los subconjuntos de atributos por cada predicado. Empero, puede mejorarse aprovechando las inferencias que pueden hacerse entre dependencias funcionales con los axiomas de Armstrong (véase Armstrong [1974]).

Epílogo

Filósofos de la ciencia, como Langley y Simon, plantean que el *descubrimiento científico equivale a la resolución de problemas*. Sustentan su hipótesis mediante programas que reproducen descubrimientos históricos, entre ellos BACON.1 y GLAUBER[36]. Dichos programas son dirigidos por datos, es decir, descubren leyes mediante métodos inductivos. Aunque sus autores los presentan con datos muy a la medida, nos muestran la manera de abordar distintos tipos de problemas mediante métodos generales de resolución, como los implementados en PRISM. La naturaleza del problema puede ser cuantitativa, como la ley de Kepler redescubierta por BACON.1, o cualitativa como la ley de reacción química redescubierta por GLAUBER.

GLAUBER redescubre que los ácidos reaccionan con álcalis para formar sales, una de las leyes de reacciones iónicas. La induce de una colección de predicados de dos tipos: el primero acerca de las cualidades de las substancias (`Has-Quality`), que considero parte del aparato perceptivo, el segundo acerca de la manera en que se relacionan conjuntos de substancias (`Reacts`). El último es el que se generaliza en función de las clases que puedan formarse en función de las cualidades.

Para analizar algunos detalles que no mencionan sus autores y realizar varios experimentos, se implementó GLAUBER con el lenguaje de programación `Haskell` a diferencia del original programado sobre el sistema para resolución de problemas PRISM(véase el apéndice A).

GLAUBER fracasó al aplicarlo al conjunto de predicados `Reacts` al que se agregaron ejemplares de reacciones entre ácidos fuertes y débiles y los predicados `Has-Quality` con las cualidades *Tastes* y *Activity* de todas las substancias (véase §2.4). El fracaso se debe a que GLAUBER procesa una cualidad a la vez, guiándose por la cualidad que aparece en más ejemplares.

Por eso SURREALIST, el método heurístico propuesto en esta tesis, tiene como tanteo principal seleccionar de todas las cualidades, únicamente las que distinguen a todas las substancias de cada ejemplar, con el supuesto

[36]A su vez se basan en el sistema resolvedor de problemas PRISM. Al hacerlo así, reafirman su hipótesis de que el descubrimiento puede efectuarse mediante métodos generales de resolución de problemas (véase capítulo 1 y Langley, Simon, Bradshaw, y Zytkow [1987]).

de que las substancias en los reactivos y los productos deben pertenecer a distintas clases de equivalencia. De este modo, al aplicar SURREALIST a dicho ejemplo, descubre más leyes, que son generalizaciones de los predicados `Reacts` con variables tomadas de clases de equivalencia formadas con las cualidades *Tastes* y *Activity*.

Se puede afirmar que SURREALIST tiene una heurística más fuerte gracias a que forma clases de equivalencia mediante *la selección de todas las cualidades que distinguen a los objetos*, en lugar de formarlas con *una cualidad que distinga a la mayoría*. Lo que brinda un poder predictivo del 100 %. Ello es posible gracias a que representa la información mediante relaciones a las que aplica operaciones del *álgebra relacional*, tal como se vio en la sección §3.2.2 y en el apéndice C.

Además de descubrir reglas y clases a partir de conjuntos de datos donde GLAUBER fracasa, SURREALIST es de uso más general, debido a que es viable su aplicación a una familia de problemas que en su planteamiento tengan dos conjuntos de predicados: `Has-Quality` con las cualidades percibidas de los objetos, y un caso más general de predicados `Reacts`, de aridad n con la forma $\texttt{Reacts}_n(s_1, \ldots, s_n)$, donde cada s_i es un conjunto cuya cardinalidad $|s_i| \in \mathbb{Z}^+$, puede ser cualquier entero positivo[37]. Asimismo, en los hechos pueden coexistir predicados `Reacts` de distinta aridad. Por ejemplo, $\texttt{Reacts}_1(s_1)$ y $\texttt{Reacts}_2(s_1, s_2)$.

SURREALIST los separará y genera distintas reglas y cúmulos de clases para cada uno. El nombre dado al predicado $\texttt{Reacts}_n$ no importa, SURREALIST puede manejar homónimos y descubrir sinónimos. Esto significa que puede generalizar las reglas aunque tengan el mismo nombre, o descubrir que dos predicados con distinto nombre relacionan de igual forma a objetos en las mismas clases.[38] Pero siempre debe respetarse el supuesto de que los objetos en la unión de todos los conjuntos relacionados pertenezcan a distintas clases.

SURREALIST también puede determinar una mejor cuantificación que GLAUBER. Como se explicó en el capítulo 2, GLAUBER puede encontrar dos cuantificaciones de la ley, una en que todo ácido reacciona con cualquier álcali para formar una sal ($\forall a \forall k \exists s(\texttt{Reacts}(\{a, k\}, \{s\}))$), o bien podría descubrir que todas las sales se forman por la reacción de un ácido con un álcali ($\forall s \exists a \exists k(\texttt{Reacts}(\{a, k\}, \{s\}))$), pero no ambas. La última cuantificación modificando la heurística para que primero seleccione las clases más pequeñas (véase Langley *et al.* [1987]). Sus autores también mencionan que no puede distinguir entre $\forall a\, \forall k\, \exists s\, (\texttt{Reacts}(\{a, k\}, \{s\}))$ y $\exists s\, \forall a\, \forall k\, (\texttt{Reacts}(\{a, k\}, \{s\}))$.

[37] En realidad esta acotado por el número de clases de equivalencia que puedan formarse con los predicados `Has-Quality`.

[38] Es posible que el nombre sea importante pero SURREALIST evita basarse en el significado que se le da en lenguaje natural.

Por su parte, SURREALIST basa su heurística para cuantificar en la detección de dependencias funcionales, lo que permite conocer las llaves. Al deducirse éstas, la cuantificación se determina con la regla que se muestra en §3.3, que encuentra una mejor cuantificación. Una aparente desventaja es la complejidad, pero la eficiencia de SURREALIST es bastante aceptable. Puede escribirse un programa especializado para formar clases en una pasada con complejidad de $O(n \log m)$, donde m es el número de clases.

En principio, la cuantificación es más costosa, ya que requiere contar todos los subconjuntos de atributos con una complejidad de $O(2^n)$ donde n es el número de atributos. Sin embargo, gracias a los los axiomas de Armstrong es posible inferir si una dependencia funcional implica a otra (véase Armstrong [1974]), lo que permite recortar el número de subconjuntos a probar: por ejemplo, si se encuentra que el atributo a_1 es una llave, no es necesario probar los demás subconjuntos que contienen a a_1, lo que se infiere del axioma de reflexividad:

$$\frac{Y \subseteq X \subseteq U}{X \to Y}.$$

por lo que una estrategia plausible consiste en comenzar con los conjuntos unitarios $\{a_1\}, \ldots, \{a_n\}$. Tampoco es necesario probar de inicio todas las tuplas; se puede comenzar con una muestra representativa evitando su aplicación a todas las tuplas, y cotejar el resultado con todos los ejemplares. La heurística puede mejorarse y no ser un problema en la práctica.

Algunas consideraciones sobre el alcance y las limitaciones de SURREALIST

Un caso a considerar corresponde a predicados de tipo `Reacts` con multiconjuntos[39] en lugar de conjuntos. En una aplicación a la ecología no es lo mismo juntar 1 hiena con 20 cebras, que juntar 1 cebra con 3 hienas; la cantidad es importante. Volviendo al ámbito de la química, las reacciones iónicas dependen del número de protones (iones de Hidrógeno, H^+) y órbitas aceptadoras de protones disponibles cuando se separa un electrólito en una solución.[40] En los ejemplos de GLAUBER todos los ácidos ceden un protón y todos los álcalis aceptan un protón. Si se combina ácido sulfúrico, H_2SO_4, con sosa, NaOH, se requieren 2 moléculas de sosa por una de ácido sulfúrico para formar sulfato de sodio, Na_2SO_4.

$$H_2SO_4 + 2\ NaOH \longrightarrow Na_2SO_4 + 2\ H_2O.$$

[39] Los multiconjuntos también llamados bolsas, son similares a los conjuntos solo que sí pueden contener repetidos los objetos, también tiene sus operaciones de intersección ($\uplus$) y unión ($\biguplus$), por ejemplo $\{\!\{a, a, b, c\}\!\} \biguplus \{\!\{a, b, d\}\!\} = \{\!\{a, a, a, b, b, c, d\}\!\}$.

[40] En una teoría química más actual, los ácidos son los donadores de protones y los álcalis los receptores de protones.

El balanceo químico brinda información para conocer la estructura de las substancias. Sin considerar a profundidad los detalles del trabajo de Mendeleyev, esta fue la base para armar la tabla periódica de los elementos, incluso prediciendo substancias desconocidas. Posteriormente, al conocer la estructura atómica se entendió el por qué de la periodicidad y de las propiedades de los elementos.

Vale la pena extender SURREALIST para considerar las cantidades. Si se pudiera hacer con multiconjuntos, tal vez permitiría inducir reglas de la *lógica difusa*[41].

Las cualidades que se perciben de los objetos pueden depender del contexto en que se observan. En el caso de las reacciones químicas, el color puede depender de la acidez de la solución, como sucede con las substancias indicadoras del pH. Una solución ingenua, consistiría en agregar un atributo para el contexto en la relación `Has-Quality`, lo que puede funcionar en algunos casos, pero las características del contexto, pueden depender de reglas y clases todavía sin descubrir. Esta circularidad no tiene una solución trivial.[42]

Es necesario explorar la posibilidad de combinar SURREALIST con métodos de la *programación lógica inductiva* para obtener las reglas de integridad de la base de datos extensional de un programa lógico y otras reglas no recursivas, así como la correcta organización de la base de datos extensional.

También se requiere investigar la elaboración de árboles de decisión, a partir de ejemplos positivos y negativos de un concepto, comparándolo con `ID3`, como se ve en la literatura de *machine learning*.

Otra posible extensión a SURREALIST, es combinar su heurística con la de BACON.1, obteniendo reglas simbólicas y numéricas.

Las siguientes tareas pueden desarrollarse a fin de continuar este trabajo:

1. Definir un procedimiento más adecuado para integrar toda la información. Actualmente el usuario aplica las operaciones siguiendo la heurística de SURREALIST a los datos.

2. Optimizar las operaciones para disminuir la complejidad del procedimiento, usando todas las optimizaciones posibles en las consultas.

3. Corregir la detección de superclases y organizarlas en CPOs.

4. Explorar la posibilidad de inferir reglas acerca de los cambios de estado como se plantea en §3.2.3.

[41] También lógica borrosa, como aparece en la literatura en lengua inglesa: *fuzzy logic*. Es una lógica multivaluada, relacionada con los multiconjuntos.

[42] El problema es análogo al problema del marco de referencia en la física.

Apéndice A

Implementación de GLAUBER

En este apéndice se describen de manera general algunas características del lenguaje `Haskell` dirigido a quienes se interesen en el código del programa. Después se expone el código de GLAUBER omitiendo los detalles del análisis sintáctico y la escritura, con el fin de concentrarnos en el procedimiento. Se describen entonces los tipos de datos, seguidos del procedimiento.

A.1 Haskell

Haskell es un lenguaje funcional puro. Un programa, llamado *script*, es una colección de definiciones de funciones en forma de ecuaciones escribiendo una ecuación para cada caso posible, el lenguaje hace concordancia de patrones. Por ejemplo para *fact*, una función definida para los naturales, hay dos casos 0 y $s(n)$, el sucesor de un natural, denotado por $(n+1)$ en `Haskell`, de modo que se escribe así:[1]

$fact\ 0$ = 1
$fact\ (n+1)$ = $(n+1) \times fact\ n$ – donde $(n+1)$ es el sucesor de n

Se prefiere definir caso por caso, aunque también es válida si se define como sigue:[2]

$fact\ n$ = **if** $n > 0$ **then** $n \times fact\ (n\text{-}1)$
else if $n = 0$ **then** 1
else $\perp$ — (**shows** n " *no es un valor correcto para fact*")

[1] En `Haskell` existen funciones más abstractas que generalizan patrones de cómputo como el anterior, un programador funcional escribiría: $fact\ n = foldr\ (\times)\ 1\ [1..n]$.

[2] En este caso en particular no son iguales. No se entrará en detalles, salvo mencionar que ambas rechazan números negativos, pero si se revisa el tipo de datos inferido por el intérprete, puede verse que son distintas. La segunda acepta números de punto flotante, mientras que la primera no. Puede forzase, por supuesto, a que sólo acepte enteros declarando la misma signatura que la primera, de modo que compute la misma función.

El orden en que se escriben las declaraciones no importa, todas quedan definidas en el ambiente. El cómputo se inicia cuando se pide evaluar una función. Si un programa va a compilarse, se define una función *main* en el módulo *Main* donde se iniciará la ejecución, de manera similar al lenguaje `C`.

En `Haskell` pueden definirse nuevos tipos de datos o también sinónimos. **data** define un tipo de datos nuevo, mientras que **type** define un sinónimo o alias, y **newtype** un tipo de datos nuevo. En el código literario, es decir, escrito con una tipografía afín al documento, normalmente se sigue la convención de nombrar a las variables de tipos de datos con letras del alfabeto griego, en texto ascii, con las primeras letras del alfabeto en minúsculas:

data α = .. o también con
type α = ..
newtype α = ..

Los tipos también pueden ser *paramétricos*, como en el siguiente ejemplo:

data *Tree* α = *NilTree* | *MkTree* (*Tree* α) (*Tree* α) | *Leaf* α
deriving (**Show**, **Eq**, **Ord**, **Read**)

Como *Tree* α es un tipo de datos nuevo, la parte **deriving** indica que hay que heredar las funciones asociadas a cada una de las clases entre paréntesis que estén defindas para los tipos en que está definido. Cuando se define un árbol de enteros, se heredarán las funciones para escribir, comparar, ordenar, y leer enteros. Un ejemplo de el uso del tipo de datos nuevo *Tree* α es:

inorder NilTree = []
inorder (*Leaf* x) = [x]
inorder (*MkTree* l r) = *inorder* l ++ *inorder* r

`Haskell` infiere el siguiente tipo:

inorder :: *Tree* $\alpha \rightarrow$ [α]

Esto significa que la función *inorder* está definida para objetos de tipo *Tree* α, devolviendo un objeto de tipo [α][3].

Cuando se desea usar otra notación para leer y escribir los tipos de datos (lo que equivale a traducir de la representación concreta a la representación abstracta y viceversa), se escribe una función analizadora de la estructura sintáctica, parser, que por lo general tendrá un nombre con la forma:

parseα :: **String** $\rightarrow$ [(α, **String**)]

El ejemplo anterior se representa de manera similar a lo que sigue:

parseTree = *parseNilTree* <|> *parseLeaf* <|> *parseMkTree*
parseNilTree = ...
parseLeaf = ...
parseMkTree = ...

[3] [α] = lista de objetos de tipo α.

Para escribir con la misma sintaxis se construye un programa *pretty-printer*, éstos tendrán un nombre con la forma:

showα :: $\alpha \rightarrow$ **String**

Tales funciones son adecuadas para *sobrecargar* las funciones de las clases **Show** α y **Read** α con las declaraciones:

instance Show α **where showsPrec** p = $(+\!+) \circ$ **show**α
instance Read α **where readsPrec** p = *determ parse*α

Las funciones analizadoras de la estructura sintáctica se escribieron usando la biblioteca de combinadores presentada en Fokker [1995]. El resultado de una función analizadora de la estructura sintáctica es una lista de las posibles estructuras encontradas, es decir una lista de éxitos. Cada estructura se devuelve junto con el resto de la cadena que falta por analizar, de ahí que las funciones tienen la signatura *parse*α :: **String** $\rightarrow$ `[(`α`,` **String**`)]`. En `Haskell` está definido el sinónimo **type ReadS** α = `[`**Char**`]` $\rightarrow$ `[(`α`,` `[`**Char**`])]`, el cual se acopla con la biblioteca de Fokker [1995][4]. Para acoplar *parse*α el resultado no debe ser ambiguo, es decir, la lista de éxitos no debe ser vacía, ni contener más de una estructura, además de que la cadena que resta por examinar debe ser vacía. En pocas palabras, debe tener la forma `[(`*Mk*α`,` `[])]` donde *Mk*α es algún valor (constructor) perteneciente al tipo α.

El módulo se importa con:

import *ReadFokker*

Para seleccionar únicamente el primer árbol encontrado se necesita la función:

takeParseTree = **fst** $\circ$ **head**

El programa está escrito en modo literario, esto significa que el código de éste capítulo, con todo y el texto es el código del programa, Haskell toma completo el texto como comentarios, y LaTeX con el paquete *Listings* lo formatea con una tipografía más estética parecida a la notación matemática más familiar.

Al final se incluye la correspondencia entre la notación tipográfica, con el texto que reconoce el intérprete de Haskell, así como algunas funciones auxiliares, de todos modos se recomienda la lectura de Fokker [1995]. Los combinadores `<*>`, `<*`, y `*>` corresponden a la yuxtaposición, mientras que el `<|>` corresponde a la alternativa, `|` en la notación BNF, por último `<@` y `<?@` permiten procesar la estructura analizada.[5]

[4] La biblioteca usa **type** *Parser* α β = `[`α`]` $\rightarrow$ `[([`α`],` β`)]` pero es fácil de modificar ya sea con una funcion que permute `([`α`],` β`)` o cambiando el código para el tipo sea **type** *Parser* α β = `[`α`]` $\rightarrow$ `[(`β`, [`α`])]`, que es lo que se hizo en el módulo *ReadFokker*.

[5] Corresponde a las rutinas semánticas en la notación BNF extendida y en las convenciones adoptados en lenguajes para escribir analizadores sintácticos como `yacc`.

Para mayor información sobre el lenguaje Haskell y programación funcional, véase Bird [2000].

A.2 Código de GLAUBER

A.2.1 Tipos de datos

Recordemos que GLAUBER no toma en cuenta la estructura química de las substancias, las fórmulas solo se usan como nombres. Sin embargo, cuando se usen éstas para nombrar substancias, dicha implementación de GLAUBER revisa que la sintaxis general de la fórmula sea correcta aunque acepta fórmulas de compuestos inexistentes como He_2S_2.

La regla para una fórmula es `<Formula> ::= (<Elem> [<Indice>])+`

Tampoco hay forma de definir sinónomos, las substancias HCl y *μriatico* son distintas para GLAUBER.

El tipo de datos *Subst*:

data *Subst* = *MkSubst* **String** | *MkNombreComun* **String**

Para las cualidades y sus valores creamos el tipo de datos *Quality* que puede ser cualquier cadena, de igual forma el valor para la cualidad está definido como el tipo de datos *Value*:

data *Quality* = *MkQuality* **String**

Los valores para las cualidades, también son un tipo de datos:

data *Value* = *MkValue* **String**

Aparte de las substancias, ya sea como fórmula o su nombre común, se tiene otro tipo de objetos, las clases. El objetivo de GLAUBER es encontrar clases. Los nombres de las clases serán substituidos por los nombres de las clases que los contienen en los distintos hechos, como son las reacciones y las cualidades:

data *Class* = *MkClass* ((*Quality*, *Value*), [*Subst*])

Para tal efecto, es necesario construir un tipo de datos `Object` que es la unión de *Subst*, *Class* y variables, aunque la última todavía no se usará en el procedimiento:

data `Object` = *ObjSubst Subst*
| *ObjClass Class*
| *ObjVar* **String**

Los hechos se representan con el tipo de datos *Fact* que es la unión de `Has-Quality`, `Reacts`, y con el fin de poder comenzar con algunas clases y leyes formadas se agregó *Class* y *Law*. Por el momento las entradas solo se darán con hechos y cualidades:

data *Fact* = *Fact*Has-Quality Has-Quality
| *Fact*Reacts Reacts
| *FactClass* *Class*
| *FactLaw* *Quantifier* Object *Fact*

La información de las reacciones se representa con el tipo de datos Reacts que relaciona a dos conjuntos de objetos, uno es *inputs* y otro *outputs*:

data Reacts = *Mk*Reacts Inputs Outputs

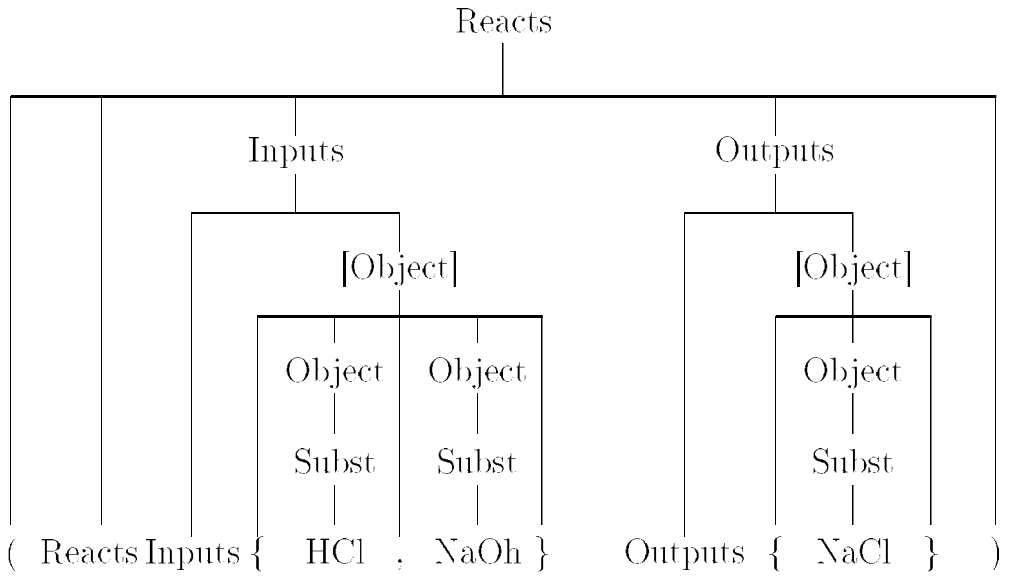
Los conjuntos de objetos, para las entradas y las salidas, están definidos así:

data Inputs = *Mk*Inputs [Object] **deriving (Eq. Ord)**
data Outputs = *Mk*Outputs [Object] **deriving (Eq. Ord)**

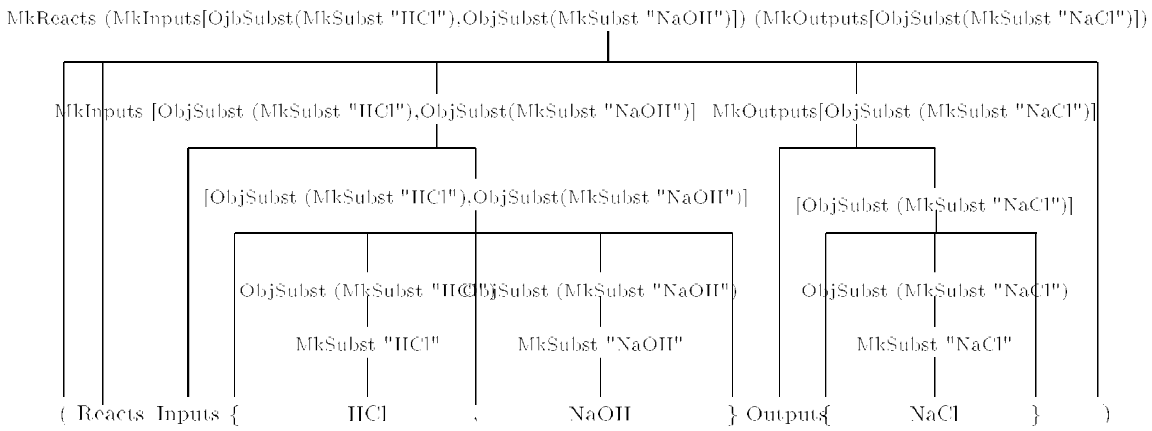
Los hechos referentes a las cualidades, se representan como sigue:
data Has-Quality = *Mk*Has-Quality Object *Quality* *Value*

Los cuantificadores usados para construir leyes son $\forall$ y $\exists$, se agregó $\exists!$ aunque GLAUBER no lo tiene:
data *Quantifier* = $\forall \mid \exists \mid \exists!$

Aunque no se muestra, todos los tipos de datos tienen definida una función $parse\alpha$ :: **[Char]** $\rightarrow$ **[(**α, **[Char])]** que efectúa el análisis de la estructura sintáctica de los objetos de tipo α. De hecho, cada tipo de datos corresponde a una categoría sintáctica. Por ejemplo al reconocer una cadena del tipo Reacts se esta construyendo el árbol de análisis sintáctico siguiente:



Al tiempo que se reconoce la estructura se van generando valores para los tipos de datos, cada expresion representa el árbol debajo de su posición:



La gramática BNF correspondiente al ejemplo es la siguiente:

<reacts> ::= "("Reacts <*inputs*> <*outputs*> ")"
<*inputs*> ::= Inputs <object-*list*>
<*outputs*> ::= Outputs <object-*list*>
<object-*list*> ::= "{" <object> ["," <object>]"}"
<object> ::= <*subst*> | <*nombre-común*>
<*subst*> ::= {<*elemento*> <*índice*> ?}
<*elemento*> ::= *H* | *He* | *Li* | *B* | ..
<*índice*> ::= "_"? <*natural*>

A.2.2 Procedimiento GLAUBER

El procedimiento GLAUBER se basa principalmente en las funciones `form-class` y `determine-quantifier`, que se aplican repetidamente a una clase candidata c sobre una tupla con el tipo ([*Fact*], [*Fact*], [*Fact*]). La entrada después del análisis sintáctico es una lista de hechos [*Fact*] que es adecuada para las funciones `form-class` y `determine-quantifier` con la función *fitFacts* :: [*Fact*] → (([*Fact*], [*Fact*], [*Fact*]), [*Class*]), la cual separa los hechos en tres entradas dependiendo de su estructura en:
(([*Fact*Reacts *fr*], [*Fact*Has-Quality *fhq*], [*FactClass fc*]), [*MkClass c*]),
rqsort :: **Ord** $\alpha \Rightarrow [\alpha] \rightarrow [\alpha]$.

Así, se ordena una lista descendentemente, y las clases tienen definida una relación de orden en base al número de elementos que contienen:

fitFacts :: [*Fact*] → (([*Fact*], [*Fact*], [*Fact*]), [*Class*])
fitFacts fs = ((*frs*, *fhqs*, *fcs*), *cs*)
where
(*frs*, *fhqs*, *fcs*) = *clasifyFacts fs*
cs = (*rqsort* (**map** (λ(*FactClass c*).*c*) *fcs* ∪ *fact*Has-Quality2*Class fhqs*))

La función *fact*Has-Quality2*Class* :: [*Fact*] → [*Class*] genera las clases posibles a partir del conjunto de los hechos *Fact*Has-Quality[6]:

[6]Las funciones *fold* se usan frecuentemente aplican un operador asociativo por la

```
fact Has-Quality2Class :: [Fact] → [Class]
 Proposito: convetir una lista de hechos FactHasQuality en una lista de clases
fact Has-Quality2Class =
    foldr (⊕) ∅ ∘ map (λ(Fact Has-Quality c).c) ∘ filter isFact Has-Quality
 where
   (Mk Has-Quality (ObjSubst s) q v) ⊕ ∅ = {MkClass ((q, v), [s])}
   (Mk Has-Quality (ObjSubst s) q v) ⊕ ((MkClass ((q', v'), subs)):cs)
     | (q, v)=(q', v') = (MkClass ((q, v), {s} ∪ subs):cs)
     | otherwise =
         (MkClass ((q', v'), subs)):((Mk Has-Quality (ObjSubst s) q v)⊕ cs)
```

La función *clasifyFacts* :: [*Fact*] → ([*Fact*], [*Fact*], [*Fact*]) es la que separa los hechos, como ya se dijo:

```
clasifyFacts :: [Fact] → ([Fact], [Fact], [Fact])
 separa los hechos en un vector ([FactReacts],[FactHasQuality],[FactClass])
clasifyFacts = foldr (⊕) ([], [], [])
  where f ⊕ (frs, fhqs, fcs)
        | isFact Reacts f = (f:frs, fhqs, fcs)
        | isFact Has-Quality f = (frs, f:fhqs, fcs)
        | isFactClass f = (frs, fhqs, f:fcs)
        | otherwise = ⊥ — ("clasifyFacts "++ shows f "no es clasificable")
```

La función:
form-class :: ([*Fact*], [*Fact*], [*Fact*]) → *Class* → ([*Fact*], [*Fact*], [*Fact*]) ,
substituye las substancias en la lista con hechos de la forma *Fact*Reacts por las clases, en este momento se trata como lista no como conjunto por lo que habrán elementos duplicados. La información se pasa a la función determine-quantifier :: *Class* → [*Fact*] → [*Fact*] que cuantificará los predicados reacts, y los devolverá en un conjunto, no una lista. A continuación se muestra el código de form-class:

```
form-class (frs, fhqs, fcs) c =
   if disjuntas c fcs ∧ parte c frs
   then (determine-quantifier c frs', fhqs', fcs')
   else (frs, fhqs, fcs)
  where
     fhqs' = ∅ ∪ map (substClassFact c) fhqs
     frs'  = map (substClassFact c) frs
     fcs'  = FactClass c:fcs  FactLaw ForAll ...
```

De las definiciones que no se muestran, *disjuntas* prueba si la clase c no tiene elementos en común con los de las clases que ya formadas, para cumplir con la restricción de que las clases tienen que ser disjuntas. *parte* prueba si una

izquierda o por la derecha y un neutro, un ejemplo del cómputo es el siguiente: **foldr** $(\oplus)$ e $[x_1, x_2, x_3] = (x_1 \oplus (x_2 \oplus (x_3 \oplus e)))$.

clase c cuando se sustituye en un predicado, no hay más de una substancia en el predicado que es elemento de la clase. *substClassFact* reemplaza a las substancias en el hecho por la clase a que pertenece.

La función `determine-quantifier` :: *Class* → `[`*Fact*`]` → `[`*Fact*`]` cuantifica los hechos contando el número de ocurrencias en la lista y en el conjunto, es decir después de eliminar duplicados. El razonamiento es el siguiente: si una clase comprende n objetos, al substituir las substancias por la clase en los predicados `Reacts`, si solo reemplaza una substancia y lo hace en n predicados, entonces el cuantificador es $\exists!$, si consideramos que no hay ninguna substancia que estuviera repetida.

En el otro extremo si al momento de eliminar duplicados la lista se reduce de k predicados a m predicados, si $m \times n = k$ quiere decir que en cada una de las m variantes se repite n, entonces el cuantificador debe ser $\forall$. En cuanto al cuantificador $\exists$, es el adecuado cuando los anteriores no se cumplen, es decir, al menos para un predicado no hay un ejemplo que cubra a todos los objetos en la clase, pero que existe alguna clase que se repite. En el programa original al cuantificador $\exists!$ no se le considera, se trata igual al cuantificador $\exists$:

```
determine-quantifier :: Class → [Fact] → [Fact]
determine-quantifier c ffs = map quantify fs'
  where quantify f =
          if (ObjClass c) ∉ (quantified f) ∧ (ObjClass c) ∈ objs f
          then (FactLaw qt (ObjClass c)) f
          else f
        quantificador c fs
          | sf×cl=cf = ∀
          | sf=cf    = ∃!  – para GLAUBER original reemplazar por ∃
          | sf<cf    = ∃
        fs  = map (substClassFact c) ffs
        fs' = ∅ ∪ fs
        sf  = length fs'
        cf  = length fs
        cl  = classLength c
        qt  = quantificador c fs
```

Observe que a diferencia del GLAUBER original, esta implementación encuentra casos donde el cuantificador es $\exists!$, con el fin de comparar, puede cambiarse a $\exists!$ por $\exists$. *quantify* solo cuantifica si el objeto no esta cuantificado y si el objeto aparece en el hecho.

Por el momento, se usará la siguiente definición de *glauber* que imprime el resultado de aplicar el procedimiento a la lista de los hechos:

```
glauber = printList ∘ formaLista ∘ uncurry (foldl form-class) ∘ fitFacts
```

A.3 Ejemplo de ejecución de *glauber*

Aunque contamos con funciones para analizar sintácticamente la entrada y escribir objetos del tipo *Fact* con la misma sintaxis con las que más adelante escribiremos un intérprete, se declarará una variable con los datos del ejemplo prestentado en Langley *et al.* [1987]:

```
facts :: [Fact]
facts  = read facts'
facts' = factsr ++newline++ factsq
factsr = "(Reacts Inputs {HCl, NaOH} Outputs {NaCl})"++newline++
         "(Reacts Inputs {HCl, KOH} Outputs {KCl})"++newline++
         "(Reacts Inputs {HNO_3, NaOH} Outputs {NaNO_3})"++newline++
         "(Reacts Inputs {HNO_3, KOH} Outputs {KNO_3})"
factsq = "(Has-Quality Object {HCl} Tastes {Sour})"++newline++
         "(Has-Quality Object {HNO_3} Tastes {Sour})"++newline++
         "(Has-Quality Object {NaCl} Tastes {Salty})"++newline++
         "(Has-Quality Object {NaNO_3} Tastes {Salty})"++newline++
         "(Has-Quality Object {KCl} Tastes {Salty})"++newline++
         "(Has-Quality Object {KNO_3} Tastes {Salty})"++newline++
         "(Has-Quality Object {NaOH} Tastes {Bitter})"++newline++
         "(Has-Quality Object {KOH} Tastes {Bitter})"
```

Si aplicamos la función *glauber* a los hechos, obtenemos:

```
Main> glauber facts
Tastes-Salty={NaCl, NaNO_3, KCl, KNO_3}
Tastes-Sour={HCl, HNO_3}
Tastes-Bitter={NaOH, KOH}
∀ x elem Tastes-Bitter (Has-Quality Object {x} Tastes{Bitter})
∀ x elem Tastes-Salty (Has-Quality Object {x} Tastes{Salty})
∀ x elem Tastes-Sour (Has-Quality Object {x} Tastes{Sour})
∀ Tastes-Bitter ∀ Tastes-Sour ∃! Tastes-Salty
    (Reacts Inputs {Tastes-Sour, Tastes-Bitter} Outputs {Tastes-Salty})
```

Para construir un intérprete que funcione como filtro, agregamos las siguientes lineas:

```
main = do s ← readFile "/dev/stdin" — lee de la entrdada estándard
          t ← (glauber ∘ read) s
          print t
          return t
```

El ejemplo anterior puede escribirse en un archivo de texto como siguie:

```
(Reacts Inputs {HCl,NaOH}   Outputs {NaCl})
(Reacts Inputs {HCl,KOH}    Outputs {KCl})
(Reacts Inputs {HNO_3,NaOH} Outputs {NaNO_3})
```

```
(Reacts Inputs {HNO_3,KOH}  Outputs {KNO_3})
(Has-Quality Object {HCl}    Tastes {Sour})
(Has-Quality Object {HNO_3}  Tastes {Sour})
(Has-Quality Object {NaCl}   Tastes {Salty})
(Has-Quality Object {NaNO_3} Tastes {Salty})
(Has-Quality Object {KCl}    Tastes {Salty})
(Has-Quality Object {KNO_3}  Tastes {Salty})
(Has-Quality Object {NaOH}   Tastes {Bitter})
(Has-Quality Object {KOH}    Tastes {Bitter})
```

Para ejecutarse en Linux como sigue:

```
$ runhugs interprete.lhs <langley.data
Tastes-Salty={NaCl,NaNO_3,KCl,KNO_3}
Tastes-Sour={HCl,HNO_3}
Tastes-Bitter={NaOH,KOH}
(Has-Quality Object {Tastes-Bitter} Tastes {Bitter})
(Has-Quality Object {Tastes-Salty} Tastes {Salty})
(Has-Quality Object {Tastes-Sour} Tastes {Sour})
ForAll Tastes-Bitter ForAll Tastes-Sour ExistsJust Tastes-Salty
(Reacts Inputs {Tastes-Sour,Tastes-Bitter} Outputs {Tastes-Salty})
()
```

El programa esta disponible, tanto en código fuente como compilado para Linux en i386 en la página: `http://www.mcc.unam.mx/~elias/tesis`

A.4 Codificación de la notación literaria

La notación literaria empleada en este apéndice se codifica en Haskell como sigue:

$\times$	*	$\perp$ — s	error s	$+\!\!+$	++
$\neq$	/=	$\oplus$	<+>	$\cup$	`union`
$=$	==	$\circ$	.	$\in$	`elem`
$\wedge$	&&	$\Rightarrow$	=>	$\notin$	`notElem`
$\vee$	||	$\rightarrow$	->	$\varnothing$	[]
$(\lambda x.e)$	(\x->e)	α	alpha	$\{a,b,c\}$	[a,b,c]
=	=	β	beta	$\forall$	ForAll
$\leftarrow$	<-	$\vdots$	$\vdots$	$\exists$	Exists
				$\exists!$	ExistsJust

Donde $xs \cup ys$ = *elimDups* $(xs +\!\!+ ys)$
where *elimDups* $\varnothing$ = $\varnothing$
elimDups $(x{:}xs)$ = $x{:}$(*elimDups*(**filter** $(x\neq)$ xs))

Apéndice B

Álgebra Relacional

La principal ventaja de SURREALIST respecto a GLAUBER es que se implementa dentro del *álgebra relacional*, lo que le permite tratar al conjunto de reacciones y al de cualidades como argumentos de operaciones, en lugar de operar con sus elementos por separado.

El *álgebra relacional* es una sistema que consta de *relaciones* y *operaciones entre relaciones.*[1] La definición de *relación* es el usual en matemáticas:un subconjunto del producto cartesiano de conjuntos llamados *dominios* $R \subseteq D_1 \times \cdots \times D_n$. Al conjunto de relaciones se le llama *base de datos.* Cada relación tiene etiquetadas sus entradas y se representa con la forma $R_i(a_1, \ldots, a_n)$, donde $R_i \subseteq X_1 \times \cdots \times X_n$. A las etiquetas $a_1, \ldots, a_n$ se les llama *atributos*, a veces, se escriben junto con sus dominios $a_1 : X_1, \ldots, a_n : X_n$. Los elementos de la relación $(x_1, \ldots, x_n) \in R_i$ se les llama *tuplas.* La forma $R_i(a_1, \ldots, a_n)$ se llama *esquema de la relación.* El conjunto de los esquemas de relación se llama *esquema de la base de datos.*

Un concepto importante es el de *dependencia funcional.* Existen dependencias funcionales entre conjuntos de atributos. Por ejemplo en la relación:

[1]En una versión extendida se agregan otros conjuntos portadores como los dominios con que se forman las relaciones, así como los números naturales. También operaciones homogéneas y heterogéneas con los dominios, los naturales y las relaciones. Una clase muy importante de dichas operaciones son las llamadas *funciones de agregados* (*aggregate functions* en inglés), que totalizan de alguna forma los valores de algún atributo en subconjuntos impropios de la relación.

Un ejemplo de esto es calcular el promedio de un valor para todas las tuplas en la relación o aglomerándolos respecto a un conjunto de valores para otros atributos.

Sqr($\overleftrightarrow{x}$, x^2)
−2 4
−1 1
0 0
1 1
2 4
3 9
4 16

Si se conoce x puede determinarse —funcionalmente— a x^2. En cambio, conociendo x^2 no puede determinarse funcionalmente a x. El valor de x, que nunca se repite, es la *llave* mientras que el valor de x^2 sí se repite. Por eso, una *llave* también se define como el conjunto de atributos que nunca se repite en la relación y por lo mismo determina funcionalmente al resto de los atributos. En la relación $R(\overleftrightarrow{x, y}, z, t)$, tanto z como t dependen funcionalmente de x y y, lo que se escribe así $\{x, y\} \to \{z, t\}$, también $\{x, y\} \to \{z\}$ y $\{x, y\} \to \{t\}$ y trivialmente $\{x, y\} \to \{x, y, z, t\}$.

B.1 Operaciones

Las operaciones del Álgebra Relacional son: proyección (π), selección (σ), juntado ($\bowtie$), división ($\div$), además de las comunes a los conjuntos ($\cup, \cap, \setminus, \times, \mathscr{P}$) entre otras. Para mayor información sobre el álgebra relacional puede consultar Maier [1983] y Ullman [1982].

π La proyección selecciona solamente unos atributos (columnas) de la relación, como las relaciones son conjuntos se eliminan tuplas repetidas. Ejemplo: sea $R(A, B, C, D) = \{(a, b, c, d), (a, d, c, b), (b, a, d, c)\}$, entonces $\pi_{A,C} R = \{(a, c), (b, d)\}$.

σ La selección solo toma las tuplas que cumplan con una condición θ de los atributos.
Ejemplo: sean $R(A, B, C, D) = \{(a, b, c, d), (a, d, c, b), (b, a, d, c)\}$ y $\theta = A =$ "a", entonces $\sigma_{A=\text{“}a\text{”}} R = \{(a, b, c, d), (a, d, c, b)\}$.

$\bowtie$ La operación de juntado toma dos relaciones con atributos en común,[2] es decir que pertenezcan a los mismos dominios, formando una que contiene a los atributos de ambas siempre se se cumpla una condición θ, la más común es la igualdad de los atributos que aparecen en los

[2]Cuando la intersección de los atributos de las relaciones es vacía, el juntado corresponde a un producto cartesiano, si $\rho(R) \cap \rho(S) = \varnothing$, donde $\rho(R(a_1, \ldots, a_n)) = \{a_1, \ldots, a_n\}$ entonces $R \bowtie S = R \times S$.

esquemas de ambas relaciones, cuando éste es el caso llama equijuntar, y no se escribe la condición θ. $R \underset{\theta}{\bowtie} S$ es equivalente a $\sigma_\theta(R \times S)$. Cuando en un equijuntado se eliminan las columnas repetidas, es decir $R \underset{\theta}{\bowtie} S = \pi_\rho(\sigma_\theta(R \times S))$, donde ρ es la unión de los atributos de R y de S, se llama *juntado natural.*
Ejemplo: Sean $R(A, B, C) = \{(1, m, 2), (1, a, 1), (2, b, 5), (3, a, 6)\}$, y $S(A, B, D) = \{(1, a, b), (2, b, c), (3, c, d)\}$, entonces $R \underset{R.A=S.A \wedge R.B=S.B}{\bowtie} S = R \bowtie S = \{(1, a, 1, 1, a, b), (2, b, 5, 2, b, c)\}$.

$\div$ La operación división[3] $S \div T = \pi_{1,2,\ldots,s-t}(S) \setminus \pi_{1,2,\ldots,s-t}((\pi_{1,2,\ldots,s-t}(S) \times T) \setminus S)$, nos permite dividir una relación S entre otra T obteniendo un cociente Q tal que si hacemos el producto cruz de Q con T obtenemos $S \setminus R$, donde R es el *residuo*) nuevamente. $S \div T = Q$, y $(Q \times T) \cup R = S$.

$\%$ El residuo[4] puede calcularse a partir de la división $S \,\%\, T = S \setminus ((S \div T) \times T)$.

El siguiente ejemplo, muestra la relación entre las operaciones $\div$ y $\%$:

S(A,B,C,D)
(a, b, c, d)
(a, b, e, f)
(b, c, e, f)
(e, d, c, d)
(e, d, e, f)
(a, b, d, e)

$\div$

T(C,D)
(c, d)
(e, f)

$=$

Q(A,B)
(a, b)
(e, d)

y $S \,\%\, T =$

R(A,B,C,D)
(b, c, e, f)
(a, b, d, e)

Q(A,B)	$\times$	T(C,D)	$\cup$	R(A,B,C,D)	$=$	S(A,B,C,D)
(a, b)		(c, d)		(b, c, e, f)		(a, b, c, d)
(e, d)		(e, f)		(a, b, d, e)		(a, b, e, f)
						(e, d, c, d)
						(e, d, e, f)
						(b, c, e, f)
						(a, b, d, e)

B.2 SQL

En el apéndice C aparece la implementación de SURREALIST, aunque de manera supervisada, en un sistema manejador de bases de datos relacionales cuyo lenguaje de programación es SQL[5], que se ha implementado en la ma-

[3] También se le llama cociente.
[4] También módulo.
[5] Abreviación de structured query language.

yoría de los SMBDRs. SQL se basa en el álgebra relacional, es un *lenguaje ortogonal*[6], la instrucción principal de SQL es:

SELECT [**ALL** | **DISTINCT** [**ON** (*expression* [, ...])]]
    * | *expression* [**AS** *output_name*] [, ...]
    [**FROM** *from_item* [, ...]]
    [**WHERE** *condition*]
    [**GROUP BY** *expression* [, ...]]
    [**HAVING** *condition* [, ...]]
    [{ **UNION** | **INTERSECT** | **EXCEPT** } [**ALL**] *select*]
    [**ORDER BY** *expression* [**ASC** | **DESC** | **USING** *operator*] [, ...]]
    [**FOR UPDATE** [**OF** *tablename* [, ...]]]
    [**LIMIT** { *count* | **ALL** }]
    [**OFFSET** *start*]

En este trabajo sólo se verá cómo pueden implementarse las operaciones del álgebra relacional con SQL. Para obtener más información puede consultar, prácticamente cualquier texto sobre bases de datos incluyendo cientos de tutoriales en Internet.

Aunque hay un estándar ANSI, hay detalles que cambian de un manejador a otro. En esta tesis solo se mostrará la traducción a la versión que tiene PostgreSQL 7.2.2, un manejador de excelente calidad que se puede bajar gratuitamente de http://www.postgresql.org.

B.2.1 Operaciones del álgebra relacional en SQL

Las operaciones del álgebra relacional pueden implementarse en SQL, aunque lo más común en un SMBD sea la traducción de SQL al álgebra relacional para su interpretación. La siguiente tabla nos muestra las operaciones vistas anteriormente:

[6]En el argot informático, se dice que un lenguaje es ortogonal cuando una instrucción puede efectuar varias funciones.

$\boldsymbol{R \cup S}$	**select** $*$ **from** R **union select** $*$ **from** S;
$\boldsymbol{R \cap S}$	**select** $*$ **from** R **intersect select** $*$ **from** S;
$\boldsymbol{R \setminus S}$	**select** $*$ **from** R **except select** $*$ **from** S;
$\boldsymbol{R \times S}$	**select** $*$ **from** R **cross join** S;
$\boldsymbol{\pi_{\rho}(R)}$	**select distinct** ρ **from** R;
$\boldsymbol{\sigma_{\theta}(R)}$	**select distinct** $*$ **from** R **where** θ;
$\boldsymbol{R \underset{\theta}{\bowtie} S}$	**select** $*$ **from** R, S **where** θ;
$\boldsymbol{R \bowtie S}$	**select** $*$ **from** R **natural join** S;
$\boldsymbol{S \div T}$	**select distinct** $*$ **from (select distinct** $1, .., s-t$ **from** S**) as** S_1 **except** **select distinct** $1, .., s-t$ **from (select distinct** $*$ **from (select distinct** $1, .., s-t$ **from** S**) as** $S1$ **cross join** T **except select distinct** $1, .., s$ **from** S**) as** S_2;
$\boldsymbol{S \,\%\, T}$	**select** $*$ **from** S **except** **select distinct** $*$ **from (select distinct** $1, .., s-t$ **from** S**) as** S_1 **except** **select distinct** $1, .., s-t$ **from (select distinct** $*$ **from (select distinct** $1, .., s-t$ **from** S**) as** S_1 **cross join** T **except select distinct** $1, .., s$ **from** S**) as** S_2;

No se ha hecho un análisis profundo de la complejidad computacional para implementar SURREALIST, ni en espacio ni en tiempo, a excepción de la división por su importancia en la heurística.

La versión mostrada en la tabla anterior corresponde a la definición formal de la operación con base a operaciones más elementales.

Implementar la división con el producto cruz es muy costoso, se requiere construir una tabla de $|R| \cdot |S|$ tuplas. En el capítulo 3 se hizo juntando las tablas y agregando un contador que nos permite separar las tuplas que pertenecen al cociente y las que pertenecen al residuo, $R \div S = \pi_{r \setminus s}(\sigma_{n=|S|}(cuenta_{r \setminus s, s}(R \bowtie S)))$ lo que a primera vista parece tener una complejidad de $O(n \log n)$.

A continuación se muestran las dos consultas. La primea es una versión un poco más eficiente que calcula solo una vez $\pi_{1,...,s-t}S$, y guarda el resultado en una tabla temporal. Aunque conviene porque simplifica la codificación, la complejidad no disminuye significativamente, a diferencia de la versión que usa junta las tablas en lugar de aplicar el producto cartesiano.

La siguiente sesión resuelve el ejemplo de la división de relaciones con el código del producto cruz simplificado:

```
elias=# CREATE TABLE S (
elias(#    A VarChar(5),
elias(#    B VarChar(5),
elias(#    C VarChar(5),
elias(#    D VarChar(5)
elias(# );
CREATE
elias=# CREATE TABLE T (C VarChar(5),D VarChar(5));
CREATE
elias=# COPY S FROM stdin DELIMITERS '&';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> a&b&c&d
>> a&b&e&f
>> b&c&e&f
>> e&d&c&d
>> e&d&e&f
>> a&b&d&e
>> \.
elias=# SELECT * FROM S;
 a | b | c | d
---+---+---+---
 a | b | c | d
 a | b | e | f
 b | c | e | f
 e | d | c | d
 e | d | e | f
 a | b | d | e
(6 rows)

elias=# COPY T FROM stdin DELIMITERS '&';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> c&d
>> e&f
>> \.
```

```
elias=# SELECT * FROM T;
 c | d
---+---
 c | d
 e | f
(2 rows)
elias=# SELECT DISTINCT *
elias-# FROM (SELECT DISTINCT A,B FROM S) AS S_1
elias-# EXCEPT
elias-# SELECT DISTINCT A,B
elias-# FROM (SELECT DISTINCT *
elias(#       FROM (SELECT DISTINCT A,B FROM S) AS S1 CROSS JOIN T
elias(#       EXCEPT SELECT * FROM S) AS S_2;
 a | b
---+---
 a | b
 e | d
(2 rows)
```

La siguiente versión, más eficiente, puede ejecutarse interactivamente de manera asistida, o escribir un programa que obtenga información de las tablas:

```
elias=# SELECT COUNT(C)AS N, A, B
elias-# FROM S NATURAL JOIN T
elias-# GROUP BY A,B;
 n | a | b
---+---+---
 2 | a | b
 1 | b | c
 2 | e | d
(3 rows)
elias=# SELECT A, B
elias-# FROM (SELECT COUNT(C)AS N, A, B
elias(#       FROM S NATURAL JOIN T
elias(#       GROUP BY A, B) AS R1
elias-# WHERE N=2;
 a | b
---+---
 a | b
 e | d
(2 rows)
```

B.3 Dependencias Funcionales

Está fuera del alcance de esta tesis profundizar en la teoría de bases de datos relacionales. Solo se mencionará el procedimiento más importante en el análisis de información y diseño del esquema de la base de datos correspondiente.

Dicho procedimiento llamado *normalización*, consiste en "simplificar" las relaciones, de tal forma que se obtiene un conjunto de esquemas de relación a los que sea imposible agregar información inconsistente con la interpretación, a menos de que se violen las restricciones impuestas por las dependencias funcionales y las referencias entre relaciones.

El resultado de cualquier operación tiene una correspondencia en el sistema modelado. Esto es, una relación que tiene una interpretación en el mundo real, por esta razón se dice que es un modelo.

En los textos de bases de datos usan otros términos, como obtener un esquema en el que se evite la pérdida de información, o se genere información espúrea.

El objetivo de la *normalización* es obtener un sistema consistente[7]. Un primer paso hacia dicho objetivo es el análisis de las dependencias funcionales (al que seguirían el análisis de dependencias multivaluadas y dependencias de juntado) para obtener la *Forma Normal de Boyce Codd* (y otras formas normales).

En la heurística usada por SURREALIST se analizan las dependencias funcionales, encontrando que hay una cuantificación correspondiente (véase §3.3).

Dado un esquema de relación $R(a_1, \ldots, a_n)$ y dos subconjuntos impropios del conjunto de atributos de la relación $X \subseteq \{a_1, \ldots, a_n\}$ y $Y \subseteq \{a_1, \ldots, a_n\}$, se dice que $X \to Y$, es decir que X determina funcionalmente a Y, o que Y depende funcionalmente de X, si $|\pi_X R(a_1, \ldots, a_n)| = |\pi_{X \cup Y} R(a_1, \ldots, a_n)|$. Es decir, en una relación r con esquema $R(a_1, \ldots, a_n)$ y con $X \to Y$, no existen dos tuplas cuyos valores para los atributos correspondientes a Y sean distintos en ambas tuplas mientras que los valores correspondientes a los atributos en X sean iguales.

Una relación $R(a_1, \ldots, a_n)$ con un conjunto de dependencias funcionales F, contiene implícitamente otras dependencias funcionales. Al conjunto de las dependencias funcionales derivables de $R(a_1, \ldots, a_n)$ y F se le llama *cerradura* o *clausura* y se escribe así F^+. Para calcular F^+ se aplica un sistema de inferencia de dependencias funcionales que se conoce como *Axiomas de Armstrong* (ver Armstrong [1974]). Con dicho sistema, dado un esquema de relación $R(a_1, \ldots, a_n)$ y un conjunto de dependencias funcionales F, es po-

[7] Lo que se calcula en el sistema es verdadero en la interpretación, es decir, su correspondencia en el mundo real.

sible computar todas las dependencias funcionales que se derivan de R y F, lo que se denomina cerradura o clausura de F y se escribe con F^+.

Apéndice C

SURREALIST con SQL

A continuación se presenta un experimento con método SURREALIST aplicado interactivamente en el intérprete de consultas `psql` de `PostgreSQL 7.2.2`[1], uno de los *sistemas manejadores de bases de datos* más avanzados. `PostgreSQL` puede extenderse con *tipos de datos* definidos por el usuario. Además extiende el *modelo relacional* con los conceptos de clases y superclases e identidad en lo que se denomina *modelo objeto-relacional*.

Aunque `SQL` (Structured Query Language) es un lenguaje "estándar" de consulta de los sistemas manejadores de bases de datos relacionales, es posible que para su ejecución en otros manejadores se requiera hacer ligeras modificaciones.

C.1 Las relaciones `Has-Quality` y `Reacts`

En esta sección se generan las relaciones con los ejemplares `Has-Quality` y `Reacts`. La relación `Has-Quality`($\overleftrightarrow{Object, Quality}, Value$), se crea como sigue:

```
create table Has-Quality (
    Object   VarChar(30),
    Quality  VarChar(30),
    Value    VarChar(30),
    primary key (Object, Quality)
);
```

Las relación `Has-Quality` se inicia con los datos del ejemplo:

```
insert into Has-Quality (Object, Quality, Value) values ('NaOH','Tastes','Bitter');
insert into Has-Quality (Object, Quality, Value) values ('KOH','Tastes','Bitter');
```

[1] El sistema es uno de los mejores; como todo lo bueno no se hizo con fines meramente comerciales, es software libre. Puede encontrar más información en `http://www.postgresql.org`.

```
insert into Has-Quality (Object, Quality, Value) values ('HCl','Tastes','Sour');
insert into Has-Quality (Object, Quality, Value) values ('HNO3','Tastes','Sour');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-O-H','Tastes','Sour');
insert into Has-Quality (Object, Quality, Value) values ('NaCl','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('KCl','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('NaNO3','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('KNO3','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-O-Na','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-O-K','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-Cl','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-NO3','Tastes','Salty');
insert into Has-Quality (Object, Quality, Value) values ('NaOH','Activity','Strong');
insert into Has-Quality (Object, Quality, Value) values ('KOH','Activity','Strong');
insert into Has-Quality (Object, Quality, Value) values ('HCl','Activity','Strong');
insert into Has-Quality (Object, Quality, Value) values ('HNO3','Activity','Strong');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-O-H','Activity','Weak');
insert into Has-Quality (Object, Quality, Value) values ('NaCl','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('KCl','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('NaNO3','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('KNO3','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-O-Na','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-O-K','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-Cl','Activity','Neutral');
insert into Has-Quality (Object, Quality, Value) values ('CH3-CO-NO3','Activity','Neutral');
```

Mostramos el contenido de `Has-Quality` mediante la consulta:

```
SELECT * FROM HasQuality;

    object     | quality  |  value
---------------+----------+---------
 NaOH          | Tastes   | Bitter
 KOH           | Tastes   | Bitter
 HCl           | Tastes   | Sour
 HNO3          | Tastes   | Sour
 CH3 -CO-O-H   | Tastes   | Sour
 NaCl          | Tastes   | Salty
 KCl           | Tastes   | Salty
 NaNO3         | Tastes   | Salty
 KNO3          | Tastes   | Salty
 CH3 -CO-O-Na  | Tastes   | Salty
 CH3 -CO-O-K   | Tastes   | Salty
 CH3 -CO-Cl    | Tastes   | Salty
 CH3 -CO-NO3   | Tastes   | Salty
 NaOH          | Activity | Strong
 KOH           | Activity | Strong
 HCl           | Activity | Strong
 HNO3          | Activity | Strong
 CH3 -CO-O-H   | Activity | Weak
 NaCl          | Activity | Neutral
 KCl           | Activity | Neutral
 NaNO3         | Activity | Neutral
 KNO3          | Activity | Neutral
 CH3 -CO-O-Na  | Activity | Neutral
 CH3 -CO-O-K   | Activity | Neutral
 CH3 -CO-Cl    | Activity | Neutral
 CH3 -CO-NO3   | Activity | Neutral
(26 rows)
```

Algunos sistemas manejadores de bases de datos como PostgreSQL permiten atributos no atómicos.[2] Esto facilita la representación de los predicados Reacts($\overleftrightarrow{Inputs, Outputs}$) mediante la tabla:

```
create table Reacts(
  Inputs VarChar(30)[],
  Outputs VarChar(30)[],
  primary key (Inputs, Outputs)[3]
);
```

E iniciarse con:

```
insert into Reacts(Inputs,Outputs) values ('{NaOH,HCl}','{NaCl}');
insert into Reacts(Inputs,Outputs) values ('{KOH,HCl}','{KCl}');
insert into Reacts(Inputs,Outputs) values ('{NaOH,HNO3}','{NaNO3}');
insert into Reacts(Inputs,Outputs) values ('{KOH,HNO3}','{KNO3}');
insert into Reacts(Inputs,Outputs) values ('{NaOH,CH3-CO-O-H}','{CH3-CO-O-Na}');
insert into Reacts(Inputs,Outputs) values ('{KOH,CH3-CO-O-H}','{CH3-CO-O-K}');
insert into Reacts(Inputs,Outputs) values ('{CH3-CO-O-H,HCl}','{CH3-CO-Cl}');
insert into Reacts(Inputs,Outputs) values ('{CH3-CO-O-H,HNO3}','{CH3-CO-NO3}');
```

Consultando el contenido de la tabla se obtiene:

```
SELECT * FROM Reacts;
       inputs        |    outputs
---------------------+---------------
 {NaOH, HCl}         | {NaCl}
 {KOH, HCl}          | {KCl}
 {NaOH, HNO3 }       | {NaNO3 }
 {KOH, HNO3 }        | {KNO3 }
 {NaOH, CH3 -CO-O-H} | {CH3 -CO-O-Na}
 {KOH, CH3 -CO-O-H}  | {CH3 -CO-O-K}
 {CH3 -CO-O-H, HCl}  | {CH3 -CO-Cl}
 {CH3 -CO-O-H, HNO3 }| {CH3 -CO-NO3 }
(8 rows)
```

También se puede ver cada elemento de los arreglos en las tablas como sigue:

```
SELECT Inputs[1] AS Inputs_1, Inputs[2]
AS Inputs_2, outputs[1]
FROM Reacts;
  inputs_1  |  inputs_2  |   outputs
------------+------------+--------------
 NaOH       | HCl        | NaCl
 KOH        | HCl        | KCl
 NaOH       | HNO3       | NaNO3
 KOH        | HNO3       | KNO3
```

[2] PostgreSQL permite definir tipos de datos, pueden definirse listas. Una alternativa presente con más frecuencia en los manejadores de bases de datos son los BLOBs (binary large objects). Así se puede guardar cualquier tipo de objeto, aunque es posible que las operaciones deban implementarse en un lenguaje anfitrión.

[3] Para incluir un arreglo en una llave se necesita definir las funciones $(=), (\leq), (\geq)$: $Arreglo \times Arreglo \rightarrow Bool$ para comparar arreglos.

```
 NaOH          | CH3 -CO-O-H| CH3 -CO-O-Na
 KOH           | CH3 -CO-O-H| CH3 -CO-O-K
 CH3 -CO-O-H| HCl           | CH3 -CO-Cl
 CH3 -CO-O-H| HNO3          | CH3 -CO-NO3
(8 rows)
```

C.2 Preparación de Reacts para formar clases

Es necesario crear la tabla ReactsN($\overleftrightarrow{N}$, *Inputs*, *Outputs*) agregando una llave de tipo entero para etiquetar cada ejemplar:

```
create table ReactsN(
    N SERIAL.[4]
    Inputs VarChar(30)[],
    Outputs VarChar(30)[],
    primary key (N)
);
```

La siguiente consulta inserta las tuplas en la relación Reacts en ReactsN. debido a que N se auto-incrementa, se numeran todas las tuplas:

```
insert into ReactsN (Inputs, Outputs)
  select Inputs, Outputs
  from Reacts;
```

Ahora se tiene una copia numerada de Reacts, como lo muestra la consulta:

```
SELECT * FROM ReactsN;
 n |        inputs         |     outputs
---+-----------------------+-----------------
 1 | {NaOH, HCl}           | {NaCl}
 2 | {KOH, HCl}            | {KCl}
 3 | {NaOH, HNO3 }         | {NaNO3 }
 4 | {KOH, HNO3 }          | {KNO3 }
 5 | {NaOH, CH3 -CO-O-H}   | {CH3 -CO-O-Na}
 6 | {KOH, CH3 -CO-O-H}    | {CH3 -CO-O-K}
 7 | {CH3 -CO-O-H, HCl}    | {CH3 -CO-Cl}
 8 | {CH3 -CO-O-H, HNO3 }  | {CH3 -CO-NO3 }
```

A continuación se desanidan los arreglos. Este ejemplo puede tratarse con consultas, pero el caso general se requiere escribir una función que devuelva el número de elementos en un arreglo,[5] para separar en subtablas homogeneas respecto al número de substancias en *Inputs* y *Outputs* y un procedimiento que desanide los conjuntos en varias tuplas con un elemento en la entrada

[4] SERIAL es un tipo entero de autoincremento de PostgreSQL, otros manejadores pueden tener algo equivalente.

[5] O mejor aún implementar un tipo de datos conjunto.

correspondiente a cada conjunto en la relación original y otra para cada objeto en los conjuntos. La siguiente consulta "desanida" *Inputs* y *Outputs*:

```
select N, 'Reacts'as Predicate, 'Inputs' as Set, Inputs[1] as Object
    into table UReacts
    from ReactsN
    union
    select N, 'Reacts'as Predicate, 'Inputs' as Set, Inputs[2] as Object
    from ReactsN
    union
    select N, 'Reacts'as Predicate, 'Outputs'as Set, Outputs[1] as Object
    from ReactsN;
```

UReacts contiene:

```
SELECT * FROM UReacts;
 n | predicate |   set   |    object
---+-----------+---------+--------------
 1 | Reacts    | Inputs  | HCl
 1 | Reacts    | Inputs  | NaOH
 1 | Reacts    | Outputs | NaCl
 2 | Reacts    | Inputs  | HCl
 2 | Reacts    | Inputs  | KOH
 2 | Reacts    | Outputs | KCl
 3 | Reacts    | Inputs  | HNO3
 3 | Reacts    | Inputs  | NaOH
 3 | Reacts    | Outputs | NaNO3
 4 | Reacts    | Inputs  | HNO3
 4 | Reacts    | Inputs  | KOH
 4 | Reacts    | Outputs | KNO3
 5 | Reacts    | Inputs  | CH3 -CO-O-H
 5 | Reacts    | Inputs  | NaOH
 5 | Reacts    | Outputs | CH3 -CO-O-Na
 6 | Reacts    | Inputs  | CH3 -CO-O-H
 6 | Reacts    | Inputs  | KOH
 6 | Reacts    | Outputs | CH3 -CO-O-K
 7 | Reacts    | Inputs  | CH3 -CO-O-H
 7 | Reacts    | Inputs  | HCl
 7 | Reacts    | Outputs | CH3 -CO-Cl
 8 | Reacts    | Inputs  | CH3 -CO-O-H
 8 | Reacts    | Inputs  | HNO3
 8 | Reacts    | Outputs | CH3 -CO-NO3
(24 rows)
```

C.3 Búsqueda de clases

A la tabla *U*Reacts pueden ligarse las cualidades como sigue:

create view *R* **as**
select *N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*
from *U*Reacts **natural join** Has-Quality
order by *N*, *Predicate*, *Set*, *Object*;

Si la relación Reacts tuviera más conjuntos, en este momento se desanidarían de la misma manera que se hizo con *Inputs* y *Outputs*. Comprobamos el contenido de las tabla *R* mediante la consulta:

```
SELECT * FROM R;
 n | predicate |   set   |    object     | quality  |  value
---+-----------+---------+---------------+----------+---------
 1 | Reacts    | Inputs  | HCl           | Activity | Strong
 1 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 1 | Reacts    | Inputs  | NaOH          | Activity | Strong
 1 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 1 | Reacts    | Outputs | NaCl          | Activity | Neutral
 1 | Reacts    | Outputs | NaCl          | Tastes   | Salty
 2 | Reacts    | Inputs  | HCl           | Activity | Strong
 2 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 2 | Reacts    | Inputs  | KOH           | Activity | Strong
 2 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 2 | Reacts    | Outputs | KCl           | Activity | Neutral
 2 | Reacts    | Outputs | KCl           | Tastes   | Salty
 3 | Reacts    | Inputs  | HNO3          | Activity | Strong
 3 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 3 | Reacts    | Inputs  | NaOH          | Activity | Strong
 3 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 3 | Reacts    | Outputs | NaNO3         | Activity | Neutral
 3 | Reacts    | Outputs | NaNO3         | Tastes   | Salty
 4 | Reacts    | Inputs  | HNO3          | Activity | Strong
 4 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 4 | Reacts    | Inputs  | KOH           | Activity | Strong
 4 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 4 | Reacts    | Outputs | KNO3          | Activity | Neutral
 4 | Reacts    | Outputs | KNO3          | Tastes   | Salty
 5 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 5 | Reacts    | Inputs  | CH3 -CO-O-H   | Tastes   | Sour
 5 | Reacts    | Inputs  | NaOH          | Activity | Strong
 5 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 5 | Reacts    | Outputs | CH3 -CO-O-Na  | Activity | Neutral
 5 | Reacts    | Outputs | CH3 -CO-O-Na  | Tastes   | Salty
 6 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 6 | Reacts    | Inputs  | CH3 -CO-O-H   | Tastes   | Sour
 6 | Reacts    | Inputs  | KOH           | Activity | Strong
 6 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 6 | Reacts    | Outputs | CH3 -CO-O-K   | Activity | Neutral
 6 | Reacts    | Outputs | CH3 -CO-O-K   | Tastes   | Salty
 7 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 7 | Reacts    | Inputs  | CH3 -CO-O-H   | Tastes   | Sour
 7 | Reacts    | Inputs  | HCl           | Activity | Strong
 7 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 7 | Reacts    | Outputs | CH3 -CO-Cl    | Activity | Neutral
 7 | Reacts    | Outputs | CH3 -CO-Cl    | Tastes   | Salty
 8 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 8 | Reacts    | Inputs  | CH3 -CO-O-H   | Tastes   | Sour
 8 | Reacts    | Inputs  | HNO3          | Activity | Strong
 8 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 8 | Reacts    | Outputs | CH3 -CO-NO3   | Activity | Neutral
 8 | Reacts    | Outputs | CH3 -CO-NO3   | Tastes   | Salty
(48 rows)
```

La siguiente consulta etiqueta las tuplas con información para separar en clases de equivalencia, y corresponde a las tuplas etiquetadas con "◁" en la página 39:

create view *Etiquetadas* **as**
select *N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*, *Label*
from *R*
natural join
(**select** *N*, *Predicate*, *Quality*, *Value*, '*Class*' **as** *Label*
from (**select** *N*, *Predicate*, *Quality*, *Value*, **count**(**all** *Object*) **as** *Class*
from *R*
group by *N*, *Predicate*, *Quality*, *Value*) **as** *R1*
where *Class*=1) **as** *Etiquetadas*
order by *N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*;

que genera la relación:

```
SELECT * FROM Etiquetadas;
 n | predicate |   set   |    object    | quality  |  value  | label
---+-----------+---------+--------------+----------+---------+-------
 1 | Reacts    | Inputs  | HCl          | Tastes   | Sour    | Class
 1 | Reacts    | Inputs  | NaOH         | Tastes   | Bitter  | Class
 1 | Reacts    | Outputs | NaCl         | Activity | Neutral | Class
 1 | Reacts    | Outputs | NaCl         | Tastes   | Salty   | Class
 2 | Reacts    | Inputs  | HCl          | Tastes   | Sour    | Class
 2 | Reacts    | Inputs  | KOH          | Tastes   | Bitter  | Class
 2 | Reacts    | Outputs | KCl          | Activity | Neutral | Class
 2 | Reacts    | Outputs | KCl          | Tastes   | Salty   | Class
 3 | Reacts    | Inputs  | HNO3         | Tastes   | Sour    | Class
 3 | Reacts    | Inputs  | NaOH         | Tastes   | Bitter  | Class
 3 | Reacts    | Outputs | NaNO3        | Activity | Neutral | Class
 3 | Reacts    | Outputs | NaNO3        | Tastes   | Salty   | Class
 4 | Reacts    | Inputs  | HNO3         | Tastes   | Sour    | Class
 4 | Reacts    | Inputs  | KOH          | Tastes   | Bitter  | Class
 4 | Reacts    | Outputs | KNO3         | Activity | Neutral | Class
 4 | Reacts    | Outputs | KNO3         | Tastes   | Salty   | Class
 5 | Reacts    | Inputs  | CH3 -CO-O-H  | Activity | Weak    | Class
 5 | Reacts    | Inputs  | CH3 -CO-O-H  | Tastes   | Sour    | Class
 5 | Reacts    | Inputs  | NaOH         | Activity | Strong  | Class
 5 | Reacts    | Inputs  | NaOH         | Tastes   | Bitter  | Class
 5 | Reacts    | Outputs | CH3 -CO-O-Na | Activity | Neutral | Class
 5 | Reacts    | Outputs | CH3 -CO-O-Na | Tastes   | Salty   | Class
 6 | Reacts    | Inputs  | CH3 -CO-O-H  | Activity | Weak    | Class
 6 | Reacts    | Inputs  | CH3 -CO-O-H  | Tastes   | Sour    | Class
 6 | Reacts    | Inputs  | KOH          | Activity | Strong  | Class
 6 | Reacts    | Inputs  | KOH          | Tastes   | Bitter  | Class
 6 | Reacts    | Outputs | CH3 -CO-O-K  | Activity | Neutral | Class
 6 | Reacts    | Outputs | CH3 -CO-O-K  | Tastes   | Salty   | Class
 7 | Reacts    | Inputs  | CH3 -CO-O-H  | Activity | Weak    | Class
 7 | Reacts    | Inputs  | HCl          | Activity | Strong  | Class
 7 | Reacts    | Outputs | CH3 -CO-Cl   | Activity | Neutral | Class
 7 | Reacts    | Outputs | CH3 -CO-Cl   | Tastes   | Salty   | Class
 8 | Reacts    | Inputs  | CH3 -CO-O-H  | Activity | Weak    | Class
 8 | Reacts    | Inputs  | HNO3         | Activity | Strong  | Class
 8 | Reacts    | Outputs | CH3 -CO-NO3  | Activity | Neutral | Class
 8 | Reacts    | Outputs | CH3 -CO-NO3  | Tastes   | Salty   | Class
(36 rows)
```

de las que se busca una muestra para comenzar con la separación de las clases. A continuación se muestra la vista $Clases_1$ seguida de la vista *Muestra*,

necesaria para separar las tuplas de la muestra:

```
create view Clases1 as
   select N, Predicate, count(Quality) as W
   from Etiquetadas
   group by N, Predicate;

create view Muestra as
   select N, Predicate, Set, Object, Quality, Value, Label
   from Etiquetadas
   natural join
   (select min(N) as N
    from Clases1
    natural join
    (select max(W) as W from Clases1) as M1) as M2;
```

La vista *Muestra* contiene las tuplas:

```
SELECT * FROM Muestra;
 n | predicate |   set   |    object    | quality  |  value  | label
---+-----------+---------+--------------+----------+---------+-------
 5 | Reacts    | Inputs  | CH3 -CO-O-H  | Activity | Weak    | Class
 5 | Reacts    | Inputs  | CH3 -CO-O-H  | Tastes   | Sour    | Class
 5 | Reacts    | Inputs  | NaOH         | Activity | Strong  | Class
 5 | Reacts    | Inputs  | NaOH         | Tastes   | Bitter  | Class
 5 | Reacts    | Outputs | CH3 -CO-O-Na | Activity | Neutral | Class
 5 | Reacts    | Outputs | CH3 -CO-O-Na | Tastes   | Salty   | Class
(6 rows)
```

La relación clases sirve para enumerar las clases, es una manera de nombrarlas:

```
create table Classes(
   Class SERIAL,
   N integer,
   Predicate TEXT,
   Set TEXT,
   Object VarChar(30),
   primary key (Class)
);
```

Al insertar las tuplas con ejemplares de las clases les asigna un número distinto:

```
insert into Classes(N, Predicate, Set, Object)
select distinct N, Predicate, Set, Object
from Muestra;
```

Se obtiene:

```
select * from classes;
 class | n | predicate |   set   |    object
-------+---+-----------+---------+--------------
     1 | 5 | Reacts    | Inputs  | CH3 -CO-O-H
     2 | 5 | Reacts    | Inputs  | NaOH
     3 | 5 | Reacts    | Outputs | CH3 -CO-O-Na
(3 rows)
```

Se necesita una relación con las cualidades y valores con que se construyen las clases, estas se obtienen con la siguiente consulta:

select *Class*, *Quality*, *Value*
into *Class* (*Class*, *Quality*, *Value*)
from *Classes* **natural join** *Muestra*;

Que produce:

```
select * From class;
 class | quality  |  value
-------+----------+---------
     1 | Tastes   | Sour
     1 | Activity | Weak
     2 | Tastes   | Bitter
     2 | Activity | Strong
     3 | Tastes   | Salty
     3 | Activity | Neutral
(6 rows)
```

Con lo anterior podemos separar todas las tuplas de la clase mediante una división. El dividendo se forma con la consutla:

create view *Dividendo* (*N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*)
as select *N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*
from *Etiquetadas*;

Con lo que el dividendo es:

```
SELECT N, Predicate, Set, Object, Quality, Value FROM Etiquetadas;
 n | predicate |   set   |    object     | quality  |  value
---+-----------+---------+---------------+----------+---------
 1 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 1 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 1 | Reacts    | Outputs | NaCl          | Activity | Neutral
 1 | Reacts    | Outputs | NaCl          | Tastes   | Salty
 2 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 2 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 2 | Reacts    | Outputs | KCl           | Activity | Neutral
 2 | Reacts    | Outputs | KCl           | Tastes   | Salty
 3 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 3 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 3 | Reacts    | Outputs | NaNO3         | Activity | Neutral
 3 | Reacts    | Outputs | NaNO3         | Tastes   | Salty
 4 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 4 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 4 | Reacts    | Outputs | KNO3          | Activity | Neutral
 4 | Reacts    | Outputs | KNO3          | Tastes   | Salty
 5 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 5 | Reacts    | Inputs  | CH3 -CO-O-H   | Tastes   | Sour
 5 | Reacts    | Inputs  | NaOH          | Activity | Strong
 5 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 5 | Reacts    | Outputs | CH3 -CO-O-Na  | Activity | Neutral
 5 | Reacts    | Outputs | CH3 -CO-O-Na  | Tastes   | Salty
 6 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 6 | Reacts    | Inputs  | CH3 -CO-O-H   | Tastes   | Sour
 6 | Reacts    | Inputs  | KOH           | Activity | Strong
 6 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 6 | Reacts    | Outputs | CH3 -CO-O-K   | Activity | Neutral
 6 | Reacts    | Outputs | CH3 -CO-O-K   | Tastes   | Salty
 7 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
```

```
 7 | Reacts    | Inputs  | HCl           | Activity | Strong
 7 | Reacts    | Outputs | CH3 -CO-Cl    | Activity | Neutral
 7 | Reacts    | Outputs | CH3 -CO-Cl    | Tastes   | Salty
 8 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 8 | Reacts    | Inputs  | HNO3          | Activity | Strong
 8 | Reacts    | Outputs | CH3 -CO-NO3   | Activity | Neutral
 8 | Reacts    | Outputs | CH3 -CO-NO3   | Tastes   | Salty
(36 rows)
```

El divisor es la relación con las clases recientemente encontradas:

create view *Divisor* (*Class*, *Quality*, *Value*)
as select *Class*, *Quality*, *Value* **from** *Class*;

Las siguiente consulta genera las reglas[6]:

create view *Reglas*
as select distinct *N*, *Predicate*, *Set*, *Class*
from (**select distinct** *Predicate*, *Set*, *Class*
from (**select** *N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*, *Class*
from *Dividendo* **natural join** *Divisor*
order by *N*, *Predicate*, *Set*, *Object*, *Quality*) **as** *R1*
natural join
(**select** ∗
from (**select** *N*, **count**(*N*) **as** *W*
from *Dividendo* **natural join** *Divisor*
group by *N*, *Predicate*) **as** *S1*
natural join
(**select max**(*W*) **as** *W*
from (**select** *N*, **count**(*N*) **as** *W*
from *Dividendo* **natural join** *Divisor*
group by *N*, *Predicate*) **as** *S*) **as** *S2*) **as** *R2*) **as** *R3*
natural join
(**select** *N*, *Class* **from** *Classes*) **as** *R4*;

Así se obtienen las reglas, aunque todavía sin cuantificar:

```
SELECT * FROM Reglas;
 n | predicate |   set   | class
---+-----------+---------+-------
 5 | Reacts    | Inputs  |     1
 5 | Reacts    | Inputs  |     2
 5 | Reacts    | Outputs |     3
(3 rows)
```

Se restan las tuplas con que se obtuvieron las reglas formando un nuevo dividendo. Si el dividendo es el conjunto vacío, la clasificación ha terminado. En caso contrario se repite el procedimiento buscando el siguiente divisor y calculando las siguientes reglas.[7]

El nuevo dividendo se obtiene mediante la consulta:

[6] Todavía sin cuantificar.

[7] Es posible que pueda fundirse el procedimiento en un solo paso, en lugar de iterar.

create view *Dividendo*$_2$
as select $*$
from *Dividendo*
where N **not in** (**select** N
from (**select** N, **count**(N) **as** W
from *Dividendo* **natural join** *Divisor*
group by N, *Predicate*) **as** *S1*
natural join
(**select max**(W) **as** W
from (**select** N, **count**(N) **as** W
from *Dividendo* **natural join** *Divisor*
group by N, *Predicate*) **as** S) **as** R);

Que tiene las tuplas:

```
SELECT * FROM Dividendo_2;
 n | predicate |   set   |    object     | quality  |  value
---+-----------+---------+---------------+----------+---------
 1 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 1 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 1 | Reacts    | Outputs | NaCl          | Activity | Neutral
 1 | Reacts    | Outputs | NaCl          | Tastes   | Salty
 2 | Reacts    | Inputs  | HCl           | Tastes   | Sour
 2 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 2 | Reacts    | Outputs | KCl           | Activity | Neutral
 2 | Reacts    | Outputs | KCl           | Tastes   | Salty
 3 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 3 | Reacts    | Inputs  | NaOH          | Tastes   | Bitter
 3 | Reacts    | Outputs | NaNO3         | Activity | Neutral
 3 | Reacts    | Outputs | NaNO3         | Tastes   | Salty
 4 | Reacts    | Inputs  | HNO3          | Tastes   | Sour
 4 | Reacts    | Inputs  | KOH           | Tastes   | Bitter
 4 | Reacts    | Outputs | KNO3          | Activity | Neutral
 4 | Reacts    | Outputs | KNO3          | Tastes   | Salty
 7 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 7 | Reacts    | Inputs  | HCl           | Activity | Strong
 7 | Reacts    | Outputs | CH3 -CO-Cl    | Activity | Neutral
 7 | Reacts    | Outputs | CH3 -CO-Cl    | Tastes   | Salty
 8 | Reacts    | Inputs  | CH3 -CO-O-H   | Activity | Weak
 8 | Reacts    | Inputs  | HNO3          | Activity | Strong
 8 | Reacts    | Outputs | CH3 -CO-NO3   | Activity | Neutral
 8 | Reacts    | Outputs | CH3 -CO-NO3   | Tastes   | Salty
(24 rows)
```

Y el nuevo divisor se obtiene, como se hizo anteriormente, buscando la tupla con mayor número de cualidades, mediante la consulta:

create view *Classes*$_2$ **as**
select N, *Predicate*, **count**(*Quality*) **as** W
from *Dividendo*$_2$
group by N, *Predicate*;

Que genera la tabla:

```
SELECT * FROM Classes_2;
 n | predicate | w
---+-----------+---
 1 | Reacts    | 4
 2 | Reacts    | 4
```

```
 3 | Reacts    | 4
 4 | Reacts    | 4
 7 | Reacts    | 4
 8 | Reacts    | 4
(6 rows)
```

De la que se selecciona una muestra, aplicando nuevamente la consulta:

create view *Muestra*$_2$ **as**
  select *N*, *Predicate*, *Set*, *Object*, *Quality*, *Value*, *Label*
  from *Etiquetadas*
  natural join
  (**select min**(*N*) **as** *N*
   from *Classes*$_2$
   natural join
   (**select max**(*W*) **as** *W* **from** *Classes*$_2$) **as** M_1) **as** M_2;

Que corresponde a las tuplas:

```
SELECT * FROM Muestra_2;
 n | predicate |   set   | object | quality  |  value  | label
---+-----------+---------+--------+----------+---------+-------
 1 | Reacts    | Inputs  | HCl    | Tastes   | Sour    | Class
 1 | Reacts    | Inputs  | NaOH   | Tastes   | Bitter  | Class
 1 | Reacts    | Outputs | NaCl   | Activity | Neutral | Class
 1 | Reacts    | Outputs | NaCl   | Tastes   | Salty   | Class
(4 rows)
```

Se agrega a la lista con las clases de equivalencia:

insert into *Classes*(*N*, *Predicate*, *Set*, *Object*)
select distinct *N*, *Predicate*, *Set*, *Object*
from *Muestra*$_2$;

Que ahora contiene:

```
SELECT * FROM Classes;
 class | n | predicate |   set   |    object
-------+---+-----------+---------+--------------
     1 | 5 | Reacts    | Inputs  | CH3 -CO-O-H
     2 | 5 | Reacts    | Inputs  | NaOH
     3 | 5 | Reacts    | Outputs | CH3 -CO-O-Na
     4 | 1 | Reacts    | Inputs  | HCl
     5 | 1 | Reacts    | Inputs  | NaOH
     6 | 1 | Reacts    | Outputs | NaCl
(6 rows)
```

Se separan las cualidades y valores que forman las clases mediante la consulta:

insert into *Class*
select *Class*, *Quality*, *Value*
from *Divisor*$_2$;

La cual tiene las cualidades:

```
SELECT * FROM Class;
 class | quality  |  value
-------+----------+---------
     1 | Tastes   | Sour
     1 | Activity | Weak
     2 | Tastes   | Bitter
```

```
      2 | Activity | Strong
      3 | Tastes   | Salty
      3 | Activity | Neutral
      4 | Tastes   | Sour
      5 | Tastes   | Bitter
      6 | Tastes   | Salty
      6 | Activity | Neutral
(10 rows)
```

Mismas que se traducen a las clases:

$C_1 = \{x \mid$ Has-Quality$(x, Tastes, Sour) \wedge$ Has-Quality$(x, Activity, Weak)\}$
$C_2 = \{x \mid$ Has-Quality$(x, Tastes, Bitter) \wedge$ Has-Quality$(x, Activity, Strong)\}$
$C_3 = \{x \mid$ Has-Quality$(x, Tastes, Salty) \wedge$ Has-Quality$(x, Activity, Neutral)\}$
$C_4 = \{x \mid$ Has-Quality$(x, Tastes, Sour)\}$
$C_5 = \{x \mid$ Has-Quality$(x, Tastes, Bitter)\}$
$C_6 = \{x \mid$ Has-Quality$(x, Tastes, Salty) \wedge$ Has-Quality$(x, Activity, Neutral)\}$

El divisor para el nuevo dividendo se obtiene como sigue:

```
create view Divisor2 (Class, Quality, Value)
as select Class, Quality, Value
from (select *
      from Classes
      where Class not in (select Class from Reglas)
     ) as R
natural join Dividendo2;
```

Que agrega las tuplas:

```
 class | quality  |  value
-------+----------+---------
     4 | Tastes   | Sour
     5 | Tastes   | Bitter
     6 | Tastes   | Salty
     6 | Activity | Neutral
(4 rows)
```

```
create view Reglas2
as select distinct Predicate, Set, Class
   from (select N, Predicate, Set, Object, Quality, Value, Class
         from Dividendo2 natural join Divisor2
         order by N, Predicate, Set, Object, Quality) as R1
   natural join
        (select *
         from (select N, count(N) as W
               from Dividendo2 natural join Divisor2
               group by N, Predicate) as S1
         natural join
              (select max(W) as W
               from (select N, count(N) as W
                     from Dividendo2 natural join Divisor2
                     group by N, Predicate) as S) as S2) as R2;
```

Esto genera las reglas:

```
SELECT * FROM Reglas_2;
 predicate |   set   | class
-----------+---------+-------
 Reacts    | Inputs  |     4
 Reacts    | Inputs  |     5
 Reacts    | Outputs |     6
(3 rows)
```

A las que corresponde la regla:

$$\texttt{Reacts}(c_4, c_5, c_6), \text{ donde } c_4 \in C_4, c_5 \in C_5, c_6 \in C_6$$

El procedimiento anterior puede optimizarse. En un trabajo futuro se deberán revisar las consultas y mediante transformaciones algebraicas fusionar los pasos de las consultas para formar clases con las de la cuantificación, en la medida en que sea posible.

Bibliografía

José Alfredo Amor Montaño. *Teoría de Conjuntos para Estudiantes de Ciencias*. Servicios Editoriales de la Facultad de Ciencias de la UNAM, México, 1997. ISBN 968-36-6582-9.

W. W. Armstrong. Dependency structures of data base relationships. En *Information Processing 74*, páginas 580–583. North-Holland, 1974.

Richard Bird. *Introducción a la Programación Funcional con Haskell*. Prentice Hall, Madrid, segunda edition, 2000.

Jeroen Fokker. Functional parsers. En Johan Jeuring y Erik Meijer, editores, *First International Spring School on Advanced Functional Programming Techniques*, volumen 925 de *Lecture Notes in Computer Science*, páginas 1–23. Springer-Verlag, Berlin Heidelberg, 1995. ISBN 3-540-59451-5.

Thomas Frank, Karl Friederick Kraiss, y Thorsten Kuhlen. Comparative analysis of fuzzy art and art-2a network clustering performance. *IEEE Transactions on Neural Networks*, 9(3):544–559, May 1998.

Peter Henderson. *Functional Programming*. Prentice Hall, 1980.

Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, segunda edición, 1973.

Pat Langley, Herbert A. Simon, G. Bradshaw, y J. Zytkow. *Scientific Discovery, Computational Explorations of the Creative Mind*. MIT Press, Cambridge, Massachusetts, 1987. ISBN 0-262-62052-9. LCCN Q175.L2443 1987. Dewey 502.8 86-10258.

Wolfgang Maass. Bounds for the computational power and learning complexity of analog neural nets (extended abstract). En *ACM Symposium on Theory of Computing*, páginas 335–344, 1993. URL `citeseer.nj.nec.com/maass92bounds.html`.

David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983. ISBN 0-914894-42-0. LCCN QA76.9.D3M33. Dewey 001.64 82-2518.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3(4):184–195, April 1960.

R. S. Michalski, Ivan Bratko, y Miroslav Kubat. *Machine Learning and Data Mining*. John Wiley, 1996.

Allen Newell y Herbert A. Simon. GPS, a program that simulates human thought. In E. A. Feigenbaum y J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 1963.

Allen Newell y Herbert A. Simon. *Human Problem Solving*. Prentice Hall, 1972.

Herbert A. Simon. The structure of ill-structured problems. *Artificial Intelligence*, 4:181–201, 1973. Reimpreso en Simon 1977.

Herbert A. Simon. *Models of Discovery*. Reidel, Dordrecht, 1977.

Herbert A. Simon. *Las Ciencias de lo Artificial*. ATE, Madrid, 1978.

Paul Thagard. *Computational Philosophy of Science*. MIT Press, Cambrige, Massachusetts, 1988. ISBN 0-262-20068-6. LCCN Q175.T479 1998. Dewey 501—dc19 87-21892.

Paul Thagard. *Conceptual Revolutions*. Princeton University Press, Princeton, New Jersey, 1992. ISBN 0-691-02490-1. LCCN Q175.T4793 1992. Dewey 501—dc20 91-20210 CIP.

Boris A. Trakhtenbrot. *Algoritmos y Computadoras*. Limusa, México, D.F., 1980.

Jeffrey David Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, segunda edición, 1982. ISBN 0-914894-36-6. LCCN QA76.9 D3U44 1983. Dewey 001.64 82-2510.

Índice alfabético

Glosario

Aridad Número de entradas (elementos) en una tupla., página 36

Extensión Manera de describir a un conjunto enumerando explícitamente sus elementos, vgr. $\{0, 1, 2\}$, página 30

Heurística De *heuriskó=descubrir, hallar*: Arte de inventar, se aplica mayoritariamente a la ideación de hipótesis y teorías científicas, así como a la resolución de problemas mediante tanteos sucesivos., página 7

Inducción El proceso de ir de un conjunto finito de hechos (un cuerpo de datos) a una generalización universalmente cuantificada (una ley)., página 11

Intensión Manera de describir un conjunto de manera constructiva, vgr. $\{x \in \mathbb{N} \ni x \leq 2\}$, página 31

Llave En una relación con esquema $R(\overleftrightarrow{a_1, \ldots, a_k}, a_{k+1}, \ldots, a_n)$, se denomina *llave* al conjunto de atributos $a_1, \ldots, a_k$, que determina funcionalmente a cada tupla. Es decir, si $R_K = \pi_{a_1,\ldots,a_k} R$, existe una función inyectiva de $R_K \to R$ y evidentemente a todas las proyecciones de R. Dado que una relación es un conjunto, siempre existe la llave trivial que se forma con todos los atributos de la relación. Por lo general la llave formada por los atributos $a_1, \ldots, a_k$ se denota resaltándolos como se ha hecho en esta tesis, colocándoles una flecha doble encima, $R(\overleftrightarrow{a_1, \ldots, a_k}, a_{k+1}, \ldots, a_n)$, o de alguna de las siguientes formas: $R(\underline{a_1, \ldots, a_k}, a_{k+1}, \ldots, a_n)$, $R(\mathbf{a_1}, \ldots, \mathbf{a_k}, a_{k+1}, \ldots, a_n)$, o algo similar, página 70

Tupla Elemento de una relación., página 36