Universidad Nacional de Rosario

Trabajo Final
Análisis de Lenguajes de Programación

Intérprete del Álgebra Relacional

Ignacio Cantore

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
6 de febrero de 2024

Índice

1. Introducción

El álgebra relacional es un lenguaje para realizar consultas sobre un modelo relacional. Este lenguaje contiene un conjunto de operaciones que toman como entrada una o dos relaciones y devuelven una relación.

Las operaciones fundamentales del álgebra relacional son: selección, proyección, renombramiento, unión, diferencia y producto cartesiano. Estas operaciones, llamadas fundamentales, son suficientes para expresar cualquier consulta.

Además, se pueden definir operaciones adicionales que se definen en términos de las operaciones fundamentales. Algunas de ellas son: intersección, unión natural, división y agrupamiento. Si bien estas operaciones no añaden potencia al álgebra relacional, permiten simplificar algunas consultas.

1.1. Objetivo

El objetivo de este trabajo es implementar un intérprete del álgebra relacional que permita definir y operar sobre relaciones, así como definir nuevas operaciones a partir de las fundamentales.

1.2. Funcionalidad

El proyecto consta de un intérprete para el álgebra relacional, el cual permite cargar relaciones (también llamadas tablas) desde archivos. A su vez, permite definir nuevas relaciones, a partir de las que haya cargadas, y nuevas operaciones, tanto desde el intérprete como desde archivos.
El intérprete permite evaluar una relación dada e imprimir en pantalla la tabla resultante.

Además, se pueden guardar tanto tablas como operadores definidos por el usuario en archivos, para ser utilizados nuevamente en el futuro.

2. Uso del programa

2.1. Compilación y ejecución

Para ejecutar el intérprete se deben tener previamente instalados Haskell y Stack. Para compilar el programa, se deben ejecutar en una terminal los siguientes comandos:

```
stack setup
stack build
```

Luego, para correrlo se debe ejecutar:

```
stack exec ReAl
```

Opcionalmente, se pueden cargar uno o más archivos haciendo:

```
stack exec ReAl file1.real file2.real
```

2.2. Operaciones

Para realizar una operación sobre una o más relaciones, basta con escribir el nombre asociado a ésta, seguido de sus argumentos. A continuación se detalla la sintaxis para cada operación fundamental:

2.2.1. Selección

Para aplicar la operación de selección sobre una relación, se debe hacer:

```
sel(cond)(R)
```

donde `cond` indica el predicado que deben satisfacer las tuplas, y `R` la relación sobre la que aplicar la operación.

El predicado debe ser una comparación: `==`, `!=`, `<`, `>`, `<=`, `>=`, que debe ser entre un atributo y una literal, o bien entre dos atributos. Éstos pueden combinarse usando los operadores lógicos `&&` (AND), || (OR) y `!` (NOT).

2.2.2. Proyección

Para aplicar la operación de proyección sobre una relación, se debe hacer:

```
proj(a1,a2,...)(R)
```

donde `a1,a2,...` indica los atributos que se desean obtener, y `R` la relación sobre la que aplicar la operación.

2.2.3. Renombramiento

Para aplicar la operación de renombramiento sobre una relación, se debe hacer:

```
rename(S)(R)
```

donde `S` indica el nuevo nombre de la relación, y `R` la relación sobre la que aplicar la operación.

2.2.4. Unión, diferencia y producto cartesiano

Para aplicar las operaciones de unión, diferencia y producto cartesiano sobre dos relaciones, se debe hacer:

```
union(R,S)
diff(R,S)
prod(R,S)
```

donde `R` y `S` indican las relaciones sobre las que aplicar la operación.

2.3. Archivos *.real*

En cada archivo se pueden dar definiciones de tablas, relaciones y operaciones.

2.3.1. Tablas

Para definir una tabla, se debe ingresar su nombre, seguido de =, y se deben poner entre {} los atributos y las filas que la compondrán. Los tipos de datos que puede tener una columa son números (enteros o *doubles*) o cadenas de caracteres. Por ejemplo:

```
R = {
a,b,c
"Rosario",30,2.5
"Pergamino",28,-3
}
```

donde la primera fila corresponde a los atributos, y los tipos de datos se infieren a partir de los valores dados en la primera tupla. Por este motivo, no se permite cargar tablas vacías.

2.3.2. Relaciones

Las relaciones se definen indicando un nombre, seguido de = y la relación que se quiere asignar. Por ejemplo:

```
T = union(R,S)
R2 = proj(c,a)(sel(b < 10)(R))
```

De este mismo modo se pueden definir relaciones en el intérprete.

2.3.3. Operaciones

Para definir una nueva operación se debe indicar su nombre, seguido de sus parámetros entre paréntesis y separados por comas. Luego, se ingresa = y la operación a definir. Por ejemplo:

```
def inter(R,S) = diff(R,diff(R,S))
def symDiff(R,S) = diff(union(R,S),inter(R,S))
```

Cabe mencionar que se pueden definir nuevas operaciones a partir de otras previamente definidas por el usuario, como se puede observar en el ejemplo. Utilizando esta misma sintaxis se pueden definir operaciones en el intérprete.

2.4. Intérprete

Al ejecutar el intérprete, se muestra un mensaje de inicio y luego el *prompt*, donde se puede comenzar a escribir comandos. Desde allí se pueden definir nuevas relaciones y operaciones, como se explicó en la sección anterior.
Por otro lado, para evaluar una operación o imprimir una tabla guardada basta con ingresar la expresión requerida; si fue posible realizar la operación, se imprimirá en pantalla la tabla resultante.
Además de esto, el intérprete cuenta con algunos comandos básicos:

- `:browse` muestra una lista de los nombres de tablas y operaciones cargadas
- `:load` permite cargar un archivo
- `:print` imprime una tabla, o la definición de un operador, en pantalla
- `:reload` recarga el último archivo que fue cargado
- `:save` permite guardar en un archivo una tabla o un operador
- `:quit` permite salir del intérprete
- `:?` o `:help` muestra una lista con los comandos disponibles, así como una descripción de su función

3. Decisiones de diseño

3.1. Representación de valores

Los tipos de valores son:

- `StringVal String`
- `IntVal Int`
- `DoubleVal Int`
- `AttrVal Name AttrName`

Se decidió tener dos tipos de valores númericos debido a que el rango de representación de los números en punto flotante es menor al de los enteros, con lo cual algunos enteros muy grandes (o muy chicos) no pueden ser representados como `Double`. Además de esto, al usar enteros se pueden evitar ciertos errores de precisión que presentan los números en punto flotante.
Adicionalmente, al *parsear* un número cuya parte decimal sea 0 (por ejemplo, 3.0), se tomó la decisión de asignarlo al tipo `IntValue`.

Por otro lado, se decidió incluir a los atributos como un tipo de valor debido a que esto permite hacer comparaciones entre éstos y los demás tipos en la operación de selección. Además, se decidió darle un parámetro adicional al constructor `AttrValue`, que corresponde al nombre de la relación de la que proviene, en caso de tener nombres de atributos repetidos (por ejemplo, al hacer un producto cartesiano).

3.2. Representación de tablas

Una tabla está representada por:

- Una lista de atributos
- Una lista de tipos
- Un conjunto de listas de valores

Se decidió usar listas ya que, si bien las relaciones son conjuntos y no debería haber un orden en sus columnas, es requerido por varias operaciones poder acceder a la i-ésima columna (es decir, al atributo, tipo y datos que la componen).

Por otro lado, se tiene una lista de tipos para que sea más fácil poder decidir si las relaciones son compatibles con la operación que se quiere realizar. Además, el tipo `Num` indica que los elementos de esa columna son de tipo númerico, independientemente de si son `Int` o `Double`, facilitando las comparaciones.

Finalmente, se decidió utilizar el tipo `Set` para las filas, ya que esto garantiza que no habrá filas repetidas.

3.3. Definición de relaciones y operaciones

Se tomó la decisión de evaluar la relación y guardar la tabla resultante a la hora de definir una nueva relación. Esto debido a que, si se guardara su definición y ésta contuviese nombres de relaciones u operaciones, al cambiar éstas también se modificaría la nueva relación.

De manera similar, a la hora de definir una nueva operación, si ésta contiene otros operadores definidos por el usuario, éstos deben resolverse y reemplazarse por su definición en términos de las operaciones fundamentales, a fin de evitar

conflictos a la hora de aplicarlos si se modifica su definición.

4. Estructuración del código

El código está formado por un `Main`, que implementa el intérprete y algunas funciones básicas, y los siguientes módulos de Haskell:

- Common.hs: contiene las definiciones de tipos
- Eval.hs: incluye las funciones que evalúan las relaciones dadas, así como las definiciones de relaciones y operaciones
- Monad.hs: contiene la mónada que se encarga de llevar el estado y lanzar errores a la hora de evaluar relaciones y hacer declaraciones
- Operations.hs: implementa las operaciones fundamentales del álgebra relacional, así como funciones auxiliares para dichas operaciones
- Parser.hs: contiene un *parser* para todas las entradas que recibe el intérprete
- PrettyPrinter.hs: incluye funciones para imprimir en pantalla tablas y operaciones definidas por el usuario, así como también para poder guardarlas en archivos

Además, en la carpeta `Examples` se incluye la definición de la operación de intersección en `intersection.real`, el cual es cargado al iniciar el intérprete. Adicionalmente, se incluye un archivo `examples.real` con algunas tablas y definiciones de nuevas relaciones a partir de éstas.

Referencias

[1] Apunte de clases: "Modelo Relacional". Claudia Deco y Cristina Bender. Teoría de Bases de Datos, Departamento de Ciencias de la Computación, UNR.

[2] Trabajos Prácticos 2 y 4. Análisis de Lenguajes de Programación, Departamento de Ciencias de la Computación, UNR.

[3] Tablas definidas en `examples.real`, tomadas de: Tablas Bancario. Teoría de Bases de Datos, Departamento de Ciencias de la Computación, UNR.

[4] Pretty printing para tablas basado en: https://stackoverflow.com/questions/5929377/format-list-output-in-haskell