Trabajo Práctico Final

Intérprete de Álgebra Relacional

Análisis de Lenguajes de Programación

Ignacio Ortego

2026

1. Introducción

Este documento describe el diseño, implementación y uso de un intérprete de Álgebra Relacional desarrollado en Haskell. El sistema permite la definición de esquemas, manipulación de relaciones mediante operadores algebraicos fundamentales y la creación de vistas, todo ello dentro de un entorno interactivo (REPL).

Se detallan las decisiones de arquitectura, el manejo de estructuras de datos inmutables y las optimizaciones implementadas.

2. Descripción del Proyecto

2.1. Objetivo y Funcionalidad

El proyecto consiste en la implementación de un motor de Álgebra Relacional con un *Read-Eval-Print Loop* (REPL) interactivo. A diferencia de un Sistema Gestor de Base de Datos (SGBD) completo basado en SQL, este software se centra en la pureza matemática del modelo relacional.

El sistema permite:

- **Carga de Datos:** Importación automática de archivos CSV convirtiéndolos en relaciones tipadas dinámicamente.
- **Evaluación Algebraica:** Ejecución de operaciones de conjuntos (Unión, Intersección, Diferencia) y operadores relacionales (Selección, Proyección, Join Natural, División, Producto Cartesiano).
- **Gestión de Vistas:** Definición de consultas con nombre que actúan como tablas virtuales reutilizables.
- **Optimización:** Un módulo dedicado reescribe el Árbol de Sintaxis Abstracta (AST) para mejorar la eficiencia antes de la ejecución.
- **Traza de Ejecución:** Visualización paso a paso del proceso de evaluación.

2.2. Motivación

La herramienta fue pensada para fines educativos y de experimentación. Utilizando el intérprete se pueden probar todas las operaciones de una manera práctica para tener una mejor comprensión de conceptos teóricos y equivalencias algebraicas. De esta forma, se podrían cargar tablas reales para comprobar el resultado de los ejercicios, aprender sobre el orden de ejecución viendo la traza, o cómo los motores SQL comerciales optimizan las consultas luego de traducirlas al Álgebra Relacional.

3. Arquitectura y Organización

El proyecto sigue una arquitectura modular estricta, separando la lógica de dominio (Core), la infraestructura de tipos (Types) y la interfaz de usuario (Interface).

```
src/
        Core/              # Nucleo de la logica de dominio
              AST.hs         # Definicion del Arbol de Sintaxis Abstracta
              Eval.hs        # Motor de evaluacion e interpretacion
              Optimizer.hs   # Reglas de reescritura y optimizacion
              Parser.hs      # Analisis sintactico (Parsec)
              Schema.hs      # Validacion y manipulacion de esquemas
        Interface/         # Capa de presentacion
              PPRel.hs       # Pretty-Printer para tablas y expresiones
              Repl.hs        # Bucle interactivo y gestion de comandos
        Types/             # Infraestructura de datos
              Common.hs      # Definicion de Relation, Tuple y Value
              Monads.hs      # Pila de Monadas (State + Error + Trace)
              RelContext.hs  # Gestion del contexto (Tablas y Vistas)
        Utils/             # Utilidades de IO
            LoadCSV.hs       # Carga de tablas en .csv
```

Listing 1: Estructura de Directorios

4. Detalles de Implementación

4.1. Representación de Datos y Esquemas

Una decisión fundamental fue el modelado de las estructuras de datos utilizando los contenedores eficientes de Haskell (`Data.Map`, `Data.Set` y `Data.Sequence`).

- **Esquema (Schema):** Se implementa como una estructura dual que combina una secuencia ordenada (`Seq`) con un conjunto (`Set`). Esta arquitectura híbrida permite preservar el orden determinista de las columnas para su visualización y salida de datos, mientras delega las operaciones de validación y manipulación de atributos al conjunto para garantizar una eficiencia de $O(\log n)$ en la búsqueda de pertenencia.
- **Tupla (Tuple):** Representada mediante un diccionario `Map String Value`, que asocia nombres de atributos con sus valores correspondientes. Esta abstracción permite un acceso logarítmico a los campos por nombre, independientemente de su posición física, facilitando la manipulación de datos y asegurando la integridad de los pares (atributo, valor).
- **Relación (Relation):** Definida como la unión de un esquema y un conjunto de tuplas (`Set Tuple`). La elección de un `Set` como estructura subyacente garantiza por construcción dos propiedades fundamentales del modelo relacional: la **unicidad de las tuplas** (eliminando automáticamente duplicados) y la **ausencia de orden intrínseco**, reflejando fielmente el concepto matemático de una relación como un subconjunto de un producto cartesiano.

4.2. Cálculo de Operaciones

El módulo `Eval.hs` transforma expresiones del AST en Relaciones resultantes. A continuación se detalla la lógica de las operaciones más complejas:

1. **Selección (σ):** Recorre el `Set` de tuplas y aplica un predicado. Gracias a la inmutabilidad, se genera un nuevo conjunto conteniendo solo las tuplas que satisfacen la condición `Columna op Valor`.
2. **Proyección (π):** Mapea cada tupla del conjunto original, construyendo nuevos `Map`s que contienen solo las claves solicitadas. Dado que al eliminar columnas pueden surgir duplicados, el uso de `Set` los colapsa automáticamente.
3. **Join Natural ($\bowtie$):** Se implementó detectando los atributos comunes entre los esquemas R y S. El algoritmo realiza un producto de los conjuntos filtrando aquellas combinaciones donde los valores de los atributos comunes coinciden, fusionando luego los mapas en una sola tupla.
4. **División ($\div$):** Implementada mediante la definición algebraica: se identifican los candidatos (valores únicos del dividendo) y se verifica, para cada uno, si al combinarlo con todas las tuplas del divisor, el resultado está presente en la relación original. Aquellos que cumplen la condición de "totalidad"forman el resultado.

4.3. Resolución Automática de Colisiones de Atributos

Durante la evaluación y optimización del Producto Cartesiano ($R \times S$), surge un desafío inherente al modelo relacional: el álgebra estricta exige que los esquemas de ambas relaciones sean disjuntos. Si ambas tablas comparten un nombre de atributo (por ejemplo, `id`), la creación de la tupla resultante generaría una colisión de claves en la estructura `Map`.

Para resolver este conflicto de manera robusta y transparente para el usuario, se implementó un algoritmo de resolución de nombres en el módulo `Schema.hs`. Mediante la función recursiva `avoidClash`, el motor detecta si un atributo de la relación derecha ya existe en el esquema acumulado de la relación izquierda. De ser así, renombra dinámicamente la columna conflictiva añadiendo un sufijo numérico secuencial (ej. `id` se transforma en `id_2`, `id_3`, etc.) hasta encontrar un identificador disponible en el conjunto de atributos (`Set String`). Esta decisión de diseño garantiza la integridad del esquema resultante sin interrumpir el flujo de evaluación de la consulta.

5. Decisiones de Diseño Importantes

5.1. Operaciones Primitivas vs. Derivadas

Una decisión de diseño clave fue implementar operaciones como el **Join Natural** y la **División** como primitivas en el código, en lugar de definirlas como composiciones de operaciones más básicas (por ejemplo, definir el Join como Producto Cartesiano + Selección + Proyección).

Justificación:

- **Eficiencia:** Derivar un Join mediante un Producto Cartesiano implica generar un conjunto intermedio de tamaño $N \times M$, para luego filtrar la inmensa mayoría de las filas. Implementarlo como primitiva permite evaluar la condición de cruce durante la generación, ahorrando memoria y tiempo de cómputo.
- **Claridad en el Error:** Al tratar estas operaciones como nativas, el sistema puede ofrecer mensajes de error semánticos en lugar de errores genéricos de proyección o selección.

- **Control de la Traza:** Permite mostrar en el historial de operaciones pasos lógicos de alto nivel (Calculando Join...) que son más significativos para el usuario que una secuencia de operaciones atómicas.

5.2. Manejo de Recursión en Vistas

Para permitir vistas que referencian a otras vistas, se implementó un sistema de control de profundidad en la mónada de estado. En lugar de mantener un grafo de dependencias complejo, el evaluador lleva un contador entero (`depth`). Cada vez que se evalúa una vista, el contador decrementa. Si llega a cero, el sistema aborta la ejecución lanzando un error de *Stack Overflow*. Esto previene eficazmente los ciclos infinitos (ej. Vista A llama a Vista B, y Vista B llama a Vista A) con una sobrecarga computacional mínima.

5.3. Importación robusta con Cassava

Para la persistencia de datos se eligió el formato CSV. Sin embargo, el parsing manual de CSV es propenso a errores (comas dentro de comillas, saltos de línea, codificación). Por eso el sistema utiliza la librería **Cassava** junto con **ByteString**.

- **Tipado Dinámico por Inferencia:** El sistema no requiere archivos de definición de esquema. Al cargar, intenta parsear cada celda como `VInt` (Entero). Si falla, recurre a `VStr` (String). Esto ofrece una buena experiencia al usuario, pudiendo incluso crear o editar tablas rápidamente en un documento de texto.

5.4. Limitaciones Intencionales del Parser

El parser de condiciones de selección se restringió a la forma `Columna == Constante`. No se permite directamente `Columna1 == Columna2` dentro de un operador `SEL`. Esta decisión simplifica el AST y fuerza al usuario a utilizar las herramientas relacionales adecuadas para comparar columnas, como el **Join Natural** (que une por igualdad de columnas homónimas) o el **Producto Cartesiano** seguido de una selección lógica, manteniendo la pureza del álgebra.

6. Manual de Uso Rápido

6.1. Ejecución

El proyecto utiliza `stack` para la gestión de dependencias y compilación.

```
# Compilar y ejecutar
stack build
stack run
```

6.2. Comandos del REPL

Una vez dentro del entorno interactivo:

Comando	Descripción
`help`	Muestra la ayuda de sintaxis.
`tablas`	Lista las tablas cargadas y vistas definidas.
`refresh`	Recarga los archivos CSV del directorio `data/`.
`VIEW x AS (exp)`	Crea una vista temporal llamada x.
`exit`	Cierra el programa.

Tabla 1: Comandos del Sistema

Las consultas se escriben directamente utilizando la sintaxis del álgebra relacional (ej. `SEL[id >5](Alumnos)`).

7. Bibliografía y Tecnologías

El desarrollo se apoyó en el ecosistema de Haskell y sus librerías estándar:

- **Parsec:** Para la construcción del parser de expresiones.
- **Containers:** Implementación eficiente de Mapas y Conjuntos.
- **Cassava & Vector:** Parsing para archivos CSV.
- **Pretty:** Formateo de salida para la visualización de tablas en consola.