Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Análisis de Lenguajes de Programación

Trabajo Final
Interprete para Algebra Relacional

Sebastián Giulianelli

Rosario, Santa Fe, Argentina

28 de octubre de 2024

Índice

1. Descripción del proyecto

AR-SQL

En el presente proyecto realicé un lenguaje de dominio específico basado en álgebra relacional llamado ar-sql. El proyecto consiste de un entorno interactivo (similar por ejemplo a un motor de bases de datos, excluyendo la parte de inserción de filas o creación de tablas) capaz de procesar consultas en este lenguaje sobre tablas de datos. Estas tablas pueden ser importadas desde motores de datos reales, específicamente en motores MySQL así como también desde archivos en formato CSV. La motivación principal del proyecto fue la de que sea una herramienta útil para complementar el aprendizaje del lenguaje algebraico proveyendo una manera sencilla de verificar los resultados obtenidos. Otra motivación quizá mas secundaria fue la de poder utilizar esta herramienta en la industria. Obviamente para lograrlo, el presente proyecto no sería mas que un punta pie inicial debido a que habría que trabajar con mayor profundidad en todos los aspectos del mismo para poder hacerlo mas robusto, eficiente, expresivo y versátil de modo que pueda asimilarse a las herramientas utilizadas hoy en día.

2. Manual de uso e instalación de software

En este proyecto se utilizó **Stack** (`https://docs.haskellstack.org/`), una herramienta sencilla para desarrollar proyectos en Haskell.

Antes que nada, puede que tengas que instalarlo. En `https://docs.haskellstack.org/en/stable/README/#how-to-install` hay guías de instalación para distintas plataformas.

Para instalar la versión correcta de GHC e instalar los paquetes necesarios basta con abrir una terminal en el directorio `AR-SQL` y ejecutar:

```
stack setup
```

Luego de esto (que puede demorar un rato), está todo listo para compilar el proyecto, haciendo:

```
stack build
```

Una vez compilado el proyecto, se puede correr el ejecutable definido en `app/Main.hs` haciendo:

```
stack exec AR-SQL
```

Esto lanzará el evaluador interactivo de AR-SQL. Con el comando :? pueden leer sobre el resto de los comandos disponibles.

3. Organización de los archivos

La estructura del proyecto es la siguiente:

```
|--- app
|    |___ Main.hs
|--- src
|    |-- Common.hs
|    |--- Eval.hs
|    |--- TypeChecker.hs
|    |--- PrettyPrinter.hs
|    |--- Error.hs
|    |___ Parse.y
|    |--- Mysql.hs
|    |--- Csv.hs
|    |___ TableOperators.y
|--- Ejemplos
|    |--- Prelude.arsql
|    |___ spj.arsql
|--- imports
|    |--- S.csv
|    |--- P.csv
|    |--- J.csv
|    |___ SPJ.csv
|--- exports
|--- README.md
|--- Setup.hs
|--- AR-SQL.cabal
|--- package.yaml
|--- stack.yaml
|___ stack.yaml.lock
```

- En el directorio `app` se define el módulo `Main`, que implementa el ejecutable final.
- En el directorio `src` se encuentran:
  - `Common` define los tipos de expresiones junto a sus ASTs y comandos presentados en la gramática junto a algunos tipos auxiliares.
  - `Eval` define los evaluadores de las expresiones y de algunos comandos.
  - `TypeChecker` define el chequeador de tipos de las expresiones.
  - `PrettyPrinter` define el printer del interprete.
  - `Error` define los mensajes de error del interprete.
  - `Parse` define el parser del lenguaje generado automáticamente por Happy.
  - `Mysql` define las funciones para importar un dataset de una base de datos MySQL.
  - `Csv` define las funciones para manejar archivos csv.
  - `TableOperators` define las operaciones sobre las relaciones basadas en el álgebra relacional.
- En el directorio `Ejemplos` está `Prelude.arsql`, con la importación de tablas csv.
  También se encuentra `spj.arsql` que utiliza las tablas importadas en el preludio para hacer operaciones sobre estas.
  (Las tablas son las que se utilizan en la materia TBD para aprender álgebra relacional, y en `spj.arsql` se muestran algunos ejercicios característicos resueltos.)
- En el directorio `imports` se encuentran las tablas importadas desde el preludio.
  (aclaración: no es necesario para importar tablas que se encuentren en la carpeta imports, basta con pasar el path adecuado al comando)
- En el directorio `exports` se encuentran las tablas exportadas desde el interprete a csv.
- El resto de los archivos son de configuración del proyecto.

4. Gramática del lenguaje

La siguiénte gramática descompuesta en su sintaxis concreta y su semántica operacional define completamente el lenguaje.

Sintaxis (concreta)	Semántica
$Comm ::=$ `'table'` Var `'='` Exp	Asignación global de tabla
| Var `'->'` Exp	Asignación local de tabla
| `'import database' '['` $ConnInfo$ `']'`	Importar una tabla en formato csv
| `'import csv'` $Path$ `'as'` Var	Importar un dataset desde un DBMS MySQL
| `'export csv'` Var `'as'` $Path$	Exportar tabla a csv
| `'drop table'` Var	Eliminar una tabla global
| `'drop operator'` Var	Eliminar un operador
| `'T' '['` $String$ `']'`	Imprimir texto
| `'operator'` Var `'=' '('` $Args$ `')' '=>'` Exp	Definir un operador
| Exp	Expresión
$Exp ::=$ `'S['` $Cond$ `']('` Exp `')'`	Operador selección (σ) en álgebra relacional
| `'P['` $Cols$ `']('` Exp `')'`	Operador proyección(π) en álgebra relacional
| `'R['` $TName$ `']('` Exp `')'`	Operador renombre(ρ) en álgebra relacional
| Exp `'*'` Exp	Producto cartesiano en álgebra relacional
| Exp `'|*|'` Exp	Producto natural en álgebra relacional
| Exp `'-'` Exp	Diferencia en álgebra relacional
| Exp `'U'` Exp	Unión en álgebra relacional
| Exp `'I'` Exp	Intersección en álgebra relacional
| Exp `'/'` Exp	División en álgebra relacional
| Var `'['` $Args$ `']'`	Aplicación de operador
| Var	Variable de tabla

$ConnInfo$	::= $ConnData$ | $ConnInfo$	información conexión a MySQL database
$ConnData$	::= `'ht:'` $String$	información de host
	| `'db:'` $String$	información de database
	| `'pt:'` Int	información de port
	| `'us:'` $String$	información de user
	| `'pw:'` $String$	información de password
$Args$	::= Var | $Args$	variables de tabla ingresadas por el usuario
$Cols$	::= Var `'|'` $Cols$ | Var	Columnas de tabla
$Cond$	::= $TermCond$	termino de condición
	| $Cond$ `'&'` $Cond$	condición AND lógico
	| $Cond$ `'|'` $Cond$	condición OR lógico
$TermCond$	::= $Atom$ `'='` $Atom$	condición de igualdad
	| $Atom$ `'<'` $Atom$	condición de menor
	| $Atom$ `'>'` $Atom$	condición de mayor
	| $Atom$ `'<='` $Atom$	condición de menor o igual
	| $Atom$ `'>='` $Atom$	condición de mayor o igual
$Atom$	::= $String$ | Int | Var	valor de celda o ingresado por el usuario
Var	::= $Text$	Nombre de variable que puede representar, tabla, operación, columna, etc

5. Decisiones de diseño

Durante el desarrollo del proyecto, hubo que tomar importantes decisiones de diseño en cada aspecto del lenguaje.

- ***Sobre la sintaxis:***

  En cuanto a la sintaxis, hubo que decidir como traducir el lenguaje del álgebra relacional puro de la forma

  $$\sigma_{\text{Cn}>200}(...) \mid \pi_{\text{C1, C2}}(...) \mid \rho_N(...) \mid \text{T1} \cup \text{T2} \mid \text{T1} \cap \text{T2} \mid \text{T1} - \text{T2} \mid \text{T1} \div \text{T2} \mid \text{T1} \times \text{T2} \mid \text{T1} \bowtie \text{T2} \mid ...$$

  a un lenguaje menos complejo en cuanto a la simbología. Es por eso que se decidió utilizar lo representado en la sintaxis concreta del lenguaje.

- ***Sobre la semántica:***

  En cuanto a la semántica, hubo que decidir algunos criterios requeridos por el álgebra relacional para saber como actuar frente a algunos escenarios. Por ejemplo, ante la repetición de un nombre de atributo para alguna operación entre relaciones, se decidió adjuntar a cada uno, el nombre de su relación de origen como criterio de denominación. Cuando no ocurre este caso, los nombres anteriormente mencionados se omiten. El criterio anterior genera ambigüedad al operar sobre relaciones con el mismo nombre, es por esto que el usuario debería ingresar un renombre de alguna de las relaciones para operar con éxito.

  Otro criterio necesario fue sobre como actuar frente al producto natural de relaciones en donde la primera contenga atributos repetidos (con diferente origen) y que en la segunda también aparezcan. Debido a este caso se decidió que el producto natural tome aquellas tuplas del producto cartesiano con todos los atributos repetidos iguales como condición.

  Otro criterio a decidir fue como actuar frente a un renombre de una relación que tenga atributos repetidos, y por ende orígenes diferentes. En este caso se decidió renombrar los orígenes de los atributos no repetidos, y dejar los orígenes de los repetidos sin modificar.

- ***Sobre los tipos:***

  En cuanto a los tipos, hubo que definir un tipo adecuado para las relaciones de modo que se pueda detectar errores previos a la evaluación de operaciones. Es por esto que se tuvieron en cuenta diferentes aspectos.

  - Nombre de la relación (Necesario para prevenir repeticiones de origen y atributos)
  - Nombre de los atributos (Necesario para prevenir atributos inexistentes)
  - Tipo de los atributos (Necesario para prevenir operaciones entre valores de distinto tipo)
  - Cantidad de atributos (Necesario para prevenir operaciones entre relaciones inválidas)
  - Atributos repetidos (Necesario para prevenir repetición de origen y de atributos)

  En base a los aspectos anteriores, se decidió optar por el siguiente tipo para las relaciones:
  `( NombreRelacion , [ ( [ RelacionOrigen ] , Atributo ) , TipoAtributo ) ] )`
  En donde la tupla `([RelacionOrigen], Atributo), TipoAtributo)` determina el tipo de un atributo, que puede ser repetido, en cuyo caso, se guardan los orígenes en la lista `[RelacionOrigen]`. (De aquí se deduce que no pueden haber atributos repetidos con distinto tipo)
  `TipoAtributo` solo permite distinguir entre números enteros y texto, (IntT y StrT)

- ***Sobre las estructuras:***

  En cuanto a las estructuras, se debía decidir sobre cual usar para representar las relaciones y sus valores de modo que puedan ser modificadas e impresas a medida que se las van operando de la manera mas simple posible. Es por esto que se definió que las filas de las relaciones se representen como una lista de listas de valores, y las relaciones como el par lista de listas (filas) y una lista de los atributos con su conjunto de ralaciones de origen agrupadas por repeticion de nombre de atributos(columnas):
  `( [[ValorCelda]] , [ ( [RelacionOrigen] , AtributoNombre ) ] )`

6. Tip para entender el código

- Teniendo en cuenta que en las decisiones de diseño siempre se habló de relaciones y atributos. Cabe aclarar que para que no haya confusiones al momento de importar/exportar tablas, a nivel de código, las relaciones son tomadas también como tablas y los atributos como sus columnas. En el código esto se ve reflejado en el nombre de las estructuras; `Table TableName, TableRow, TableValue`, etc.

7. Bibliografía

1. Fueron utilizados como guías estructurales del proyecto los trabajos prácticos 3 y 4 realizados durante el cursado de la materia.

2. `https://docs.haskellstack.org/en/stable/README/#how-to-install`

3. `https://hackage.haskell.org/package/mysql-haskell`

4. `https://hackage.haskell.org/package/cassava-0.5.3.2/docs/Data-Csv.html`

5. `https://haskell-happy.readthedocs.io/en/latest/using.html`