SEMANTICS-BASED RAPID PROTOTYPING OF A SUBSET OF SQL

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria

Joint work with William Steingartner, Department of Computers and Informatics, Technical University of Košice, Slovakia. Supported by Aktion Österreich–Slowakei project 2024-05-15-001 and KEGA project 030TUKE-4/2023.

JOHANNES KEPLER UNIVERSITY LINZ

SubSQL: A Subset of SQL

SQL (Structured Query Language): a language for managing relational databases.

- The SQL data definition commands `CREATE TABLE` and `DROP TABLE`.
- The SQL data manipulation commands `INSERT INTO`, `UPDATE`, and `DELETE`.
- The SQL data query command `SELECT` with `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`.
- Subqueries, i.e., `SELECT` queries nested inside other `SELECT` queries (at arbitrary depth) where inner queries may refer to variables from outer queries.
- The SQL aggregate functions `MIN()`, `MAX()`, `COUNT()`, `SUM()` and `AVG()` that reduce a table column to a single value.
- The combination of tables by `CROSS JOIN` (short: `","`), `(INNER) JOIN`, `LEFT (OUTER) JOIN`, `RIGHT (OUTER) JOIN`, `FULL (OUTER) JOIN`.
- Values of the integer type `INT(`n`)` and the string type `VARCHAR(`n`)` (where the parameter n, however, does actually not limit the size of the value but only the width of its display).
- `NULL` values and a 3-valued logic that includes the truth value `NULL` (interpreted as “unknown”).

Semantics-based implementation of a rapid prototype of SubSQL in SLANG.

SubSQL: Syntax

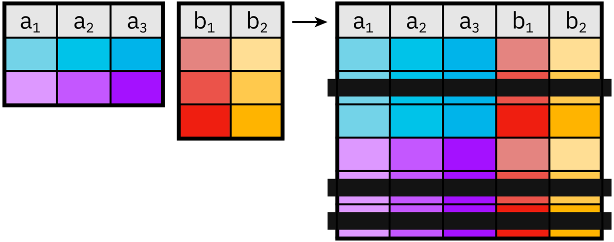
```
CREATE TABLE Persons (
    PersonId INT(6) PRIMARY KEY,
    LastName VARCHAR(255) NOT NULL,
    FirstName VARCHAR(255),
    Address VARCHAR(255) DEFAULT "(none)",
    City VARCHAR(255));

INSERT INTO Persons VALUES (1, "Schreiner", "Wolfgang", "Landstr. 1", "Linz");
UPDATE Persons SET PersonId = PersonId+1, Address="None" WHERE City <> "Linz" OR Address = "";
DELETE FROM Persons WHERE FirstName = "Wolfgang" OR PersonId > 3 AND City < "Linz";

SELECT P1.PersonId,P2.PersonId
  FROM Persons P1 JOIN Persons P2 ON (P1.PersonId=P2.PersonId)
  WHERE P1.FirstName = "Wolfgang" AND P2.PersonId=0
  GROUP BY P1.FirstName ORDER BY P1.PersonId;
SELECT P.PersonId,P.City FROM Persons P
  WHERE P.PersonId > (SELECT AVG(PersonId) FROM Persons WHERE City = P.City);

DROP TABLE Persons;
```

SubSQL Semantics



https://commons.wikimedia.org/wiki/File:Relational_Algebra_Join.svg

Based on tables, rows, columns, cells, their combination and manipulation.

Semantic Domains

SubSQL is given a denotational semantics that is based on the following domain.

$$
\begin{aligned}
\mathit{State} &:= \mathit{store}\colon \mathit{Store} \times \mathit{db}\colon \mathit{Database} \times \mathit{tables}\colon \mathit{Tables} \times \mathit{rows}\colon \mathit{Table} \\
\mathit{Store} &:= \mathit{Id} \to_{\perp} \mathit{Value} \\
\mathit{Database} &:= \mathit{Id} \to_{\perp} \mathit{Table} \\
\mathit{Tables} &:= \mathit{Table}^{*} \\
\mathit{Table} &:= \mathit{Row}^{*} \\
\mathit{Row} &:= \mathit{Value}^{*}
\end{aligned}
$$

The semantic value $v = [\![p]\!](s, \ldots)$ of a syntactic phrase p depends on a given "current state" $s \in \mathit{State}$.

The Meaning of Column References

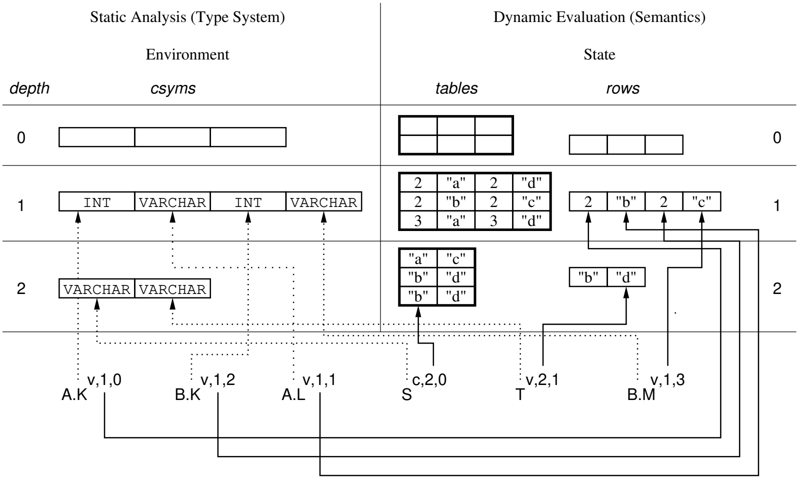
The meaning of a column reference $CRef$ is derived from the following information:

- $column$: a binary value that indicates whether the identifier refers to an entire table column (which can be determined from the *tables* stack) or just to a single value in a column (which can be determined from the *rows* stack).
- $depth$: the nesting depth at which the identifier has been introduced; it therefore also represents the stack depth at which the identifier can find its value (from the *tables* stack or from the *rows* stack).
- pos: the column position denoted by the identifier within the referenced table/row.

The type checker annotates every column reference as $CRef^{\langle column, depth, pos \rangle}$ which is used by the evaluator to determine its semantics.

Static Analysis and Semantic Evaluation

```
CREATE TABLE A (K INT(3), L VARCHAR(5)); CREATE TABLE B (K INT(3), M VARCHAR(8)); CREATE TABLE C (S VARCHAR(5), T VARCHAR(8));
```




```
... (SELECT * FROM A,B WHERE A.K = B.K AND A.L = (SELECT MIN(S) FROM C WHERE T = B.M)) ...
```

The Meaning of Column References: Static Analysis

The environment captures the information about the syntactic context of a phrase.

$$
\begin{aligned}
&\mathit{Env} := \mathit{db}\colon \mathit{TableEnv} \times \mathit{venv}\colon \mathit{VarEnv} \times \mathit{tenv}\colon \mathit{TableEnv} \times \\
&\quad \mathit{cstack}\colon \mathit{ColumnSymbols}^* \times \mathit{cenv}\colon \mathit{ColumnEnv} \times \\
&\quad \mathit{ckind}\colon (\mathit{ColumnSymbol} \to \mathit{Kind}) \times \mathit{cdepth}\colon (\mathit{ColumnSymbol} \to \mathbb{N}) \times \\
&\quad \mathit{depth}\colon N \times \mathit{gsyms}\colon \mathit{CoumnSymbolSet}
\end{aligned}
$$

```
Env ⊢ Exp: exp(ExpType)
=======================
env ⊢ ColumnRef: columnRef(csym)
type = type(csym.type)  kind = env.ckind(csym)  depth = env.cdepth(csym)
pos ∈ ℕ  env.cstack[depth][pos] = csym
grouped = depth < length(env.cstack)-1 ∨ csym ∈ env.gsyms
etype = ⟨type:type, kind:kind, group:grouped⟩
-----------------------------
env ⊢ ColumnRef^{csym,pos,kind,depth}: exp(etype)
```

The typechecker determines the annotation of CRef from the current environment.

The Meaning of Column References: SLANG Specification

```
judgement #Env# ⊢ Exp: exp(#ExpType#)
{
  inference env ⊢ ExpRef[ColumnRef,Annotation]: exp(etype)
  {
    env ⊢ ColumnRef: columnRef(csym);
    kind: #Kind# = #env.ckind().get(csym)#;
    depth: #Integer# = #env.cdepth().get(csym)#;
    pos: #Integer# = #env.cstack().get(depth).list().indexOf(csym)#;
    type: #Type# = #csym.ctype().type()#;
    # boolean grouped = depth < env.cstack().size()-1 ||
        env.gsyms().contains(csym); #
    etype = #new ExpType(type, kind, grouped)#;
    # Annotation.setColumnInfo(csym,pos,kind,depth); #
  }
}
```

The Meaning of Column References: Dynamic Evaluation

The semantics of an expression is a mapping from states to values.

```
⟦Exp⟧: State →⊥ Value
======================
⟦ColumnRef^{csym,pos,kind,depth}⟧(s) :=
  match kind with
    Kind.cell → s.rows[depth][pos]
  | Kind.column →
      Value.seq(λi∈domain(s.tables[depth]). s.tables[depth][i][pos])
```

The evaluator computes from the annotation the value of $CRef$ in the current state.

The Meaning of Column References: SLANG Specification

```
function ⟦Exp⟧: #State# → #Value#
{
  equation ⟦ExpRef[ColumnRef,Annotation]⟧(s) = v
  {
    # Integer pos = Annotation.getColumnPos();
      Kind kind = Annotation.getColumnKind();
      Integer depth = Annotation.getColumnDepth(); #
    v = #switch (kind) {
           case Kind.cell -> s.rows().list().get(depth).list().get(pos);
           case Kind.column -> {
             List<Value> values = new ArrayList<Value>();
             for (Row row : s.currentTable().list())
               values.add(row.list().get(pos));
             yield new Value.Seq(new Values(values));
            }
          }#;
  }
}
```

The SELECT Expression: Static Analysis

```
Env ⊢ SelectExp: selectExp(ColumnSymbols)
==========================================
env0 ⊢ FromClause: fromClause(env1)
env1 ⊢ WhereClause: whereClause
env1 ⊢ GroupByClause: groupByClause(env2)
env2 ⊢ HavingClause: havingClause
env2 ⊢ SelectClause: selectClause(env3,csyms)
env3 ⊢ orderByClause
---------------------------------
env0 ⊢
  "SELECT" DistinctClause SelectClause FromClause WhereClause
   GroupByClause HavingClause OrderByClause ";": selectExp(csyms)
```

The SELECT Expression: SLANG Specification

```
judgement #Env# ⊢ SelectExp: selectExp(#ColumnSymbols#)
{
  inference env0 ⊢ TheSelectExp[DistinctClause,SelectClause,FromClause,
    WhereClause,GroupByClause,HavingClause,OrderByClause]: selectExp(csyms)
  {
    env0 ⊢ FromClause: fromClause(env1);
    env1 ⊢ WhereClause: whereClause;
    env1 ⊢ GroupByClause: groupByClause(env2);
    env2 ⊢ HavingClause: havingClause;
    env2 ⊢ SelectClause: selectClause(env3,csyms);
    env3 ⊢ OrderByClause: orderByClause;
  }
}
```

The SELECT Expression: Dynamic Evaluation

```
⟦SelectExp⟧: State →⊥ Table
============================
⟦"SELECT" DistinctClause SelectClause FromClause^csyms0 WhereClause
          GroupByClause HavingClause OrderByClause ";"⟧(s) :=
  let table1 = ⟦FromClause⟧(s) in
  let s0 = newTable(s,table1) in
  let table2 = ⟦WhereClause⟧(s0,table1) in
  let tables3 = ⟦GroupClause⟧(table2) in
  let tables4 = ⟦HavingClause⟧(s0,tables3)) in
  let table5 = ⟦SelectClause⟧(s0,tables4) in
  let table6 = ⟦DistinctClause⟧(allRows,table5) in
  let table7 = ⟦OrderByClause⟧(table6) in
  table7
```

The SELECT Expression: SLANG Specification

```
function ⟦SelectExp⟧: #State# → #Table#
{
  equation ⟦TheSelectExp[DistinctClause,SelectClause,FromClause,
    WhereClause,GroupByClause,HavingClause,OrderByClause]⟧(s) = table
  {
    table1 = ⟦FromClause⟧(s);
    s0: #State# = #s.newTable(table1)#;
    table2 = ⟦WhereClause⟧(s0,table1);
    tables3 = ⟦GroupByClause⟧(table2);
    tables4 = ⟦HavingClause⟧(s0,tables3);
    table5 = ⟦SelectClause⟧(s0,tables4);
    allRows: #Function<Table,Table># = #(Table t)->t.allRows()#;
    table6 = ⟦DistinctClause⟧(allRows,table5);
    table = ⟦OrderByClause⟧(table6);
  }
}
```

The SubSQL Implementation in SLANG

Starting point: a semi-formal definition of SubSQL in 2,400 plain text lines.

- $\rightsquigarrow$ 3,500 SLANG lines (120 KB) $\rightsquigarrow$ 18,600 generated Java lines (590 KB)
  - Syntax domains: 100 SLANG lines.
  - Type system: 1,400 SLANG lines.
  - Semantics: 1,100 SLANG lines.
  - Printer: 400 SLANG lines.
  - Parser: 400 SLANG lines.
- Mathematical domains/operations: 1,300 Java lines.
- Persistent database: 500 Java lines.
- SubSQL Java API: 150 Java lines.

`https://www.risc.jku.at/research/formal/software/SLANG/public/SubSQL.tgz`

Wolfgang Schreiner and William Steingartner, *Semantics-Based Rapid Prototyping of a Subset of SQL*, RISC Report 25-02, doi:10.35011/risc.25-02.