Database Languages and their Compilers

Prof. Dr. Torsten Grust

Database Systems Research Group
U Tübingen

Winter 2010

2 A Query Calculus for SQL

- Although SQL comes with a quite limited number of syntactic forms, we can express a **broad diversity of query types**:
  - **Projection**

    $$\texttt{SELECT}\ \ A, B$$
    $$\texttt{FROM}\ \ S$$

  - **Aggregation**

    $$\texttt{SELECT}\ \ \texttt{MAX}(A)$$
    $$\texttt{FROM}\ \ S$$

  - **Quantification**

    $$\texttt{FORALL}\ x\ \texttt{IN}\ S : x\ \texttt{<>}\ 0$$

  - $+$ **selection**, **joins**, **grouping**, **sorting**, ...

- The techniques presented in this chapter try to find a **uniform representation** for this diversity found in SQL.

2.1 SQL's Nested Loops Semantics

- SQL's operation can largely be understood by re-expressing queries via **nested iteration** (deeper nesting occurs with joins, grouping, ...):
  - **Projection**

```
SELECT  A, B
  FROM  S
```

$$
\begin{array}{l}
c \leftarrow \varnothing; \\
\textsc{foreach}\ x \in S\ \textsc{do} \\
\quad c \leftarrow c \uplus \{(x.A, x.B)\}; \\
\textsc{return}\ c;
\end{array}
$$

  - **Aggregation**

```
SELECT  MAX(A)
  FROM  S
```

$$
\begin{array}{l}
c \leftarrow -\infty; \\
\textsc{foreach}\ x \in S\ \textsc{do} \\
\quad c \leftarrow \max(c, x.A); \\
\textsc{return}\ c;
\end{array}
$$

■ Quantification

$$\texttt{FORALL}\ x\ \texttt{IN}\ S : x\ \texttt{<>}\ 0$$

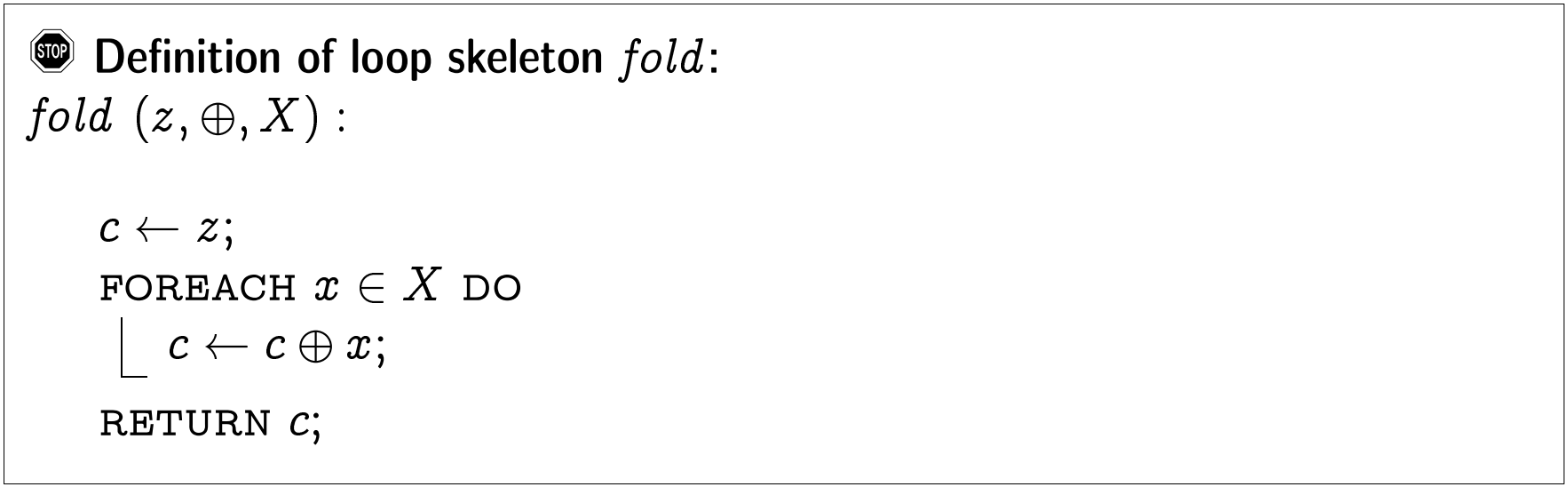
> ✎ **How would the appropriate loop look like in this case?**

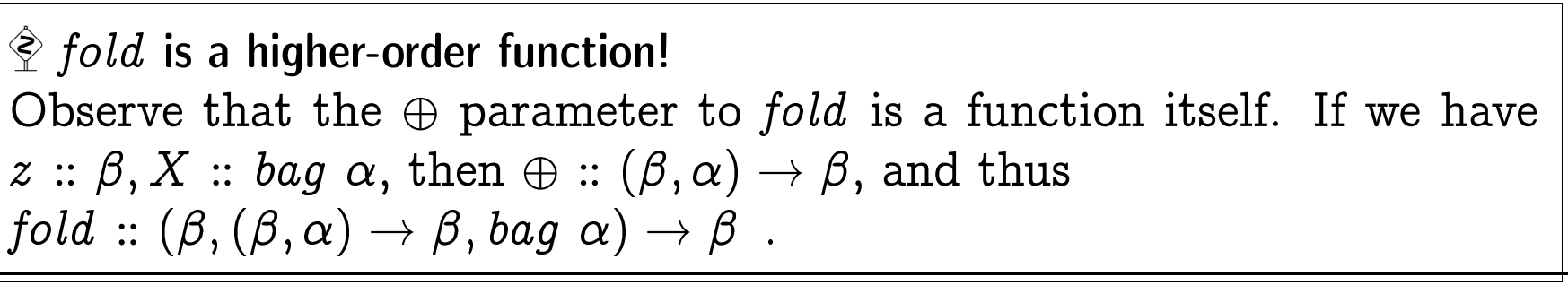
▶ Apparently, it's merely the **initializer** for c and the **accumulating function** that make a difference here:

query type	initializer	accumulator
Projection	$\varnothing$	$\uplus$
Aggregation (MAX)	$-\infty$	max
⋮	⋮	⋮
Quantification (FORALL)	*true*	$\wedge$
⋮	⋮	⋮

2.2 *fold*

- It seems worth to try to map the SQL diversity onto a **single loop skeleton**, parameterized by the initializer z and accumulator function $\oplus$:





- The simple query types are readily expressed via *fold* if we **plug-in suitable definition for z and $\oplus$**:

  - **Projection**

$$\begin{array}{rl} \texttt{SELECT} & A, B \\ \texttt{FROM} & S \end{array} \qquad \begin{array}{l} fold\ (\varnothing, \oplus, S) \\ \text{with}\ \oplus(c, x) = c \uplus \{(x.A, x.B)\} \end{array}$$

  - **Aggregation**

$$\begin{array}{rl} \texttt{SELECT} & \texttt{MAX}(A) \\ \texttt{FROM} & S \end{array} \qquad \begin{array}{l} fold\ (-\infty, \oplus, S) \\ \text{with}\ \oplus(c, x) = \max(c, x.A) \end{array}$$

  - **Quantification**

$$\texttt{FORALL}\ x\ \texttt{IN}\ S : x\ \texttt{<>}\ 0 \qquad \begin{array}{l} fold\ (true, \oplus, S) \\ \text{with}\ \oplus(c, x) = c \wedge (x \neq 0) \end{array}$$

- This idea extends quite naturally to more complex types of SQL queries and arbitrary levels of query nesting.
  - A **join** between to tables leads to nested *fold*s of depth 2:

$$\begin{array}{rl}\texttt{SELECT} & x.A, y.B \\ \texttt{FROM} & R\ x, S\ y\end{array} \qquad \begin{array}{l} \mathit{fold}\ (\varnothing, \oplus, R) \\ \text{with}\ \oplus(c, x) = \begin{array}{l} c \uplus \mathit{fold}\ (\varnothing, \otimes, S) \\ \quad \text{with}\ \otimes(c', y) = c' \uplus \{(x.A, y.B)\} \end{array}\end{array}$$

  - We can also handle **selection** as illustrated below:

$$\begin{array}{rl}\texttt{SELECT} & A \\ \texttt{FROM} & S \\ \texttt{WHERE} & A > B\end{array} \qquad \begin{array}{l} \mathit{fold}\ (\varnothing, \oplus, S) \\ \text{with}\ \oplus(c, x) = c \uplus (\mathit{if}\ (x.A > x.B)\ \{x.A\}\ \mathit{else}\ \varnothing)\end{array}$$

> **The role of $\varnothing$, $-\infty$, true, ...**
>
> $$c \uplus \varnothing = c \qquad \max(c, -\infty) = c \qquad c \wedge \mathit{true} = c$$

- It turns out that one can embrace **all** SQL constructs via nested cascades of *fold*s.
  - One last example:

$$\begin{array}{rl}\texttt{SELECT} & \texttt{COUNT}(*) \\ \texttt{FROM} & R\ x \\ \texttt{WHERE} & \texttt{EXISTS}\ y\ \texttt{IN}\ S : x.A = y.B\end{array}$$

$$\begin{array}{l} \mathit{fold}\ (0, \oplus, \mathit{fold}\ (\varnothing, \otimes, R)) \\ \text{with}\quad \oplus(c, x) = c + 1 \\ \qquad\quad \otimes(c, x) = c \uplus (\mathit{if}\ (\mathit{fold}\ (\mathit{false}, \odot, S))\ \{x\}\ \mathit{else}\ \varnothing) \\ \qquad\qquad\qquad \text{with}\ \odot(c, y) = c \vee (x.A = y.B) \end{array}$$

✎ **Give the *fold* equivalent for the SQL query shown here:**

$$\begin{array}{rll}\texttt{SELECT} & \texttt{SUM}(x) & \\ \texttt{FROM} & R\ x, & \left(\begin{array}{rl}\texttt{SELECT} & y \\ \texttt{FROM} & S\ y \\ \texttt{WHERE} & x.A = y.B\end{array}\right)\end{array}$$

- The resulting *fold* programs look rather forbidding, though. Explicit introduction of accumulators, variables, and their nested scopes make this representation hardly readable for a human.

2.3 The Monoid Comprehension Calculus

- There is, however, a way to eat the cake and have it, too:

  We will now introduce a different representation for SQL queries, **the Monoid Comprehension Calculus (MCC)** [1, 2], with the following characteristics:

  ① SQL queries can be mapped to MCC via a straightforward **parse tree matching** mechanism;
  ② MCC is **human readable** and amenable to simple **analysis** techniques (variable scoping, type checking);
  ③ MCC has a **formal semantics**, completely based on *fold*.

- The core of MCC are **comprehension** expressions:

$$[\, e \mid q_1, \ldots, q_n \,]^M$$

  - the **qualifiers** q_i are either **generators** $v_i \leftarrow e_i$ (v_i variable) or **predicates**;
  - e, the **head**, is an arbitrary expression;
  - a variable bound in a generator q_i is visible in qualifiers $q_{i+1}, \ldots, q_n$ and in e;
  - M denotes a monoid.

- **Example** (the bag of all x in $X :: bag\ \mathbb{N}$ so that $x \neq 42$):

$$[\, x \mid x \leftarrow X, x \neq 42 \,]^{bag}$$

  (here: head x, qualifiers $q_1 : x \leftarrow X, q_2 : x \neq 42$).

  All bindings for x that pass the predicate are accumulated using the given monoid's binary operation (here: $\uplus$) to form the result.

- A comprehension over a monoid M can obviously express **joins**, **selection**, and **projection** over collections ($M \in \{bag, set, list\}$).

- Assume monoid $all = (true, \wedge)$, then we can express universal **quantification** in MCC, e.g. (with $X :: bag\ \alpha, Y :: bag\ \beta$ and α, β comparable):

$$[\, x > y \mid x \leftarrow X, y \leftarrow Y \,]^{all}$$

- Analogously, comprehensions may denote different types of **aggregations** as they are common in SQL (using monoids $sum = (0, +), min = (\infty, \min), \ldots$).

> ✎ `COUNT` **in MCC**
> How could we express the SQL `COUNT` aggregation, as in
>
> ```
> SELECT  COUNT(*)
>   FROM  R x
>  WHERE  x > 0
> ```
>
> in MCC?

- As variable binding and scoping in MCC and SQL coincide, it's rather simple to map SQL queries to equivalent MCC expressions.
  - SQL formulation:

$$\begin{array}{rl} \texttt{SELECT} & \texttt{COUNT}(*) \\ \texttt{FROM} & R\ x \\ \texttt{WHERE} & \texttt{EXISTS}\ y\ \texttt{IN}\ S : x.A = y.B \end{array}$$

MCC equivalent ($some = (false, \vee)$):

$$\left[\, 1 \mid x \leftarrow R, \left[\, x.A = y.B \mid y \leftarrow S \,\right]^{some} \,\right]^{sum}$$

> ✎ **How does the MCC equivalent look like here?**
>
> $$\begin{array}{rl} \texttt{SELECT} & \texttt{DISTINCT}\ x, y \\ \texttt{FROM} & R\ x, S\ y \\ \texttt{WHERE} & x.A = y.A\ \texttt{AND}\ \texttt{FORALL}\ z\ \texttt{IN}\ T : x.A > z.A \end{array}$$

- In preparation for the development of a systematic mapping, let us list all monoids we will need to express SQL constructs in MCC (the purpose of $lift^M$ will become clear in a minute):

M	carrier	$lift^M$	z^M	$\oplus^M$
bag	$bag\ \alpha$	$\{\cdot\}$	$\varnothing$	$\uplus$
set	$set\ \alpha$	$\{\cdot\}$	$\varnothing$	$\cup$
$list$	$list\ \alpha$	$[\cdot]$	$[\,]$	$+\!\!+$
all	$\mathbb{B}$	id	$true$	$\wedge$
$some$	$\mathbb{B}$	id	$false$	$\vee$
sum	$\mathbb{N}$	id	0	$+$
max	α	id	$-\infty$	max
min	α	id	∞	min

N.B.

- $id(x) = x$, $\{\cdot\}(x) = \{x\}$, $[\cdot](x) = [x]$;
- $+\!\!+$ is list concatenation, e.g. $[1,2,3]+\!\!+[4] = [1,2,3,4]$.

- Earlier we have said that comprehensions have a **semantics based on the *fold* loop skeleton**.
  - The three equations below reveal comprehensions to be mere syntactic sugar for (nested) *fold*s:

$$[\, e \mid \,]^M = lift^M(e) \tag{1}$$

$$[\, e \mid v_1 \leftarrow e_1, q \,]^M = fold(z^M, \otimes, e_1) \quad \text{with } \otimes(c, v_1) = c \oplus^M [\, e \mid q \,]^M \tag{2}$$

$$[\, e \mid p, q \,]^M = if\ (p)\ [\, e \mid q \,]^M\ else\ z^M \tag{3}$$

✎ **Derive the *fold* form of this SQL query via MCC (SQL ⇢ MCC ⇢ *fold*):**

```
SELECT  COUNT(*)
  FROM  R x
 WHERE  EXISTS y IN S : x.A = y.B
```

2.4 Mapping SQL to MCC

- In what follows we will develop a detailed mapping, say $\mathbb{T}$, for the complete subset of SQL constructs to MCC.

- For any given SQL query q, $\mathbb{T}(q)$ will be an equivalent MCC expression.
  - $\mathbb{T}$ will be a **syntax-directed mapping**, i.e., for each syntactic construct of SQL we will define which MCC expression $\mathbb{T}$ shall emit.
  - As the SQL parser provides us with a parse tree for query q, $\mathbb{T}$ will essentially be a **tree pattern matcher**, e.g.

$$\mathbb{T}\left(\begin{array}{ccc} & \mathit{expr} & \\ \mathit{expr} & \langle + \rangle & \mathit{expr} \\ e_1 & & e_2 \end{array}\right) \quad = \quad \mathbb{T}(e_1) + \mathbb{T}(e_2)$$

  - To save space, we will abbreviate the above and write

$$\mathbb{T}(e_1 + e_2) \quad = \quad \mathbb{T}(e_1) + \mathbb{T}(e_2) \ .$$

- We shall make use of a small number of simple **helper functions** which will be introduced as soon as we need them. The two most important of these are:
  - $\mathtt{rv}$ (derive MCC variable name from SQL fragment and/or index i):

$$\begin{array}{lcl} \mathtt{rv}(i, e\ \mathtt{AS}\ \langle \mathtt{ID}, v \rangle) & = & v \\ \mathtt{rv}(i, \langle \mathtt{ID}, v \rangle) & = & v \\ \mathtt{rv}(i, e) & = & R_i \end{array}$$

    We will use $\mathtt{rv}$ to generate unique MCC variable names in situations of lax SQL variable usage:

$$\begin{array}{l} \mathtt{SELECT}\ \ \ldots \\ \quad \mathtt{FROM}\ \ \underbrace{\mathtt{T}}_{\mathtt{rv(1,T)}=\mathtt{T}_1}\ ,\ \underbrace{\mathtt{S\ AS\ s}}_{\mathtt{rv(2,S\ AS\ s)}=\mathtt{s}}\ ,\ \underbrace{\mathtt{T}}_{\mathtt{rv(3,T)}=\mathtt{T}_3}\ ,\ \underbrace{(\mathtt{SELECT} \ldots \mathtt{FROM})}_{\mathtt{rv(4,SELECT...\ )}=R_4} \end{array}$$

  - $\mathtt{col}$ (derive column name from SQL fragment and/or index i):

$$\begin{array}{lcl} \mathtt{col}(i, e\ \mathtt{AS}\ \langle \mathtt{ID}, c \rangle) & = & c \\ \mathtt{col}(i, \langle \mathtt{ID}, c \rangle) & = & c \\ \mathtt{col}(i, \langle \mathtt{ID}, v \rangle . \langle \mathtt{ID}, c \rangle) & = & v{\cdot}c \\ \mathtt{col}(i, e) & = & X_i \end{array}$$

(a) **Basic** `SELECT–FROM` **block**:

$$\mathbb{T}\begin{pmatrix}\texttt{SELECT} & p_1,\ldots,p_n\\ \texttt{FROM} & e_1,\ldots,e_m\end{pmatrix} = \mathbb{T}\begin{pmatrix}\texttt{SELECT} & p_1,\ldots,p_n\\ \texttt{FROM} & e_1,\ldots,e_m\\ \texttt{WHERE} & \texttt{TRUE}\end{pmatrix}$$

(b) **Basic** `SELECT–FROM–WHERE` **block**:

$$\mathbb{T}\begin{pmatrix}\texttt{SELECT} & p_1,\ldots,p_n\\ \texttt{FROM} & e_1,\ldots,e_m\\ \texttt{WHERE} & q\end{pmatrix} =$$

$$\begin{aligned}&[\,(\mathtt{col}(1,p_1) : \mathbb{T}(p_1),\\ &\quad\ldots,\\ &\quad\mathtt{col}(n,p_n) : \mathbb{T}(p_n))\mid\\ &\qquad\mathtt{rv}(1,e_1) \leftarrow \mathbb{T}(e_1),\ldots,\mathtt{rv}(m,e_m) \leftarrow \mathbb{T}(e_m),\mathbb{T}(q),\\ &\qquad * \equiv \mathtt{rv}(1,e_1).* \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathtt{rv}(m,e_m).*\,]^{bag}\end{aligned}$$

(c) **Basic** `SELECT–FROM–WHERE` **block with** `DISTINCT`:

$$\mathbb{T}\left(\begin{array}{rl}\texttt{SELECT DISTINCT} & p_1,\ldots,p_n\\ \texttt{FROM} & e_1,\ldots,e_m\\ \texttt{WHERE} & q\end{array}\right) =$$

$$\begin{array}{l}[\,(\mathbb{col}(1,p_1) : \mathbb{T}(p_1),\\ \quad \ldots,\\ \quad \mathbb{col}(n,p_n) : \mathbb{T}(p_n)) \mid\\ \quad \mathbb{rv}(1,e_1) \leftarrow \mathbb{T}(e_1),\ldots,\mathbb{rv}(m,e_m) \leftarrow \mathbb{T}(e_m),\mathbb{T}(q),\\ \quad * \equiv \mathbb{rv}(1,e_1).* \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathbb{rv}(m,e_m).*\,]^{set}\end{array}$$

- In (b) and (c) we have used operator $+\!\!+$ to glue together two tuples:

$$(a_1 : e_1,\ldots,a_n : e_n) +\!\!+ (b_1 : e'_1,\ldots,b_m : e'_m) = (a_1 : e_1,\ldots,a_n : e_n, b_1 : e'_1,\ldots,b_m : e'_m)$$

- We use $v \equiv e'$ to denote local named definitions in comprehensions:

$$[\,e \mid q_1,\ldots,v \equiv e', q_{i+1},\ldots,q_n\,]^M \quad = \quad [\,e[e'/v] \mid q_1,\ldots,q_{i+1}[e'/v],\ldots,q_n[e'/v]\,]^M$$

(d) `VALUES` **clause**:

$$\mathbb{T}(\texttt{VALUES}\ e_1, \ldots, e_n) =$$
$$\{\mathbb{T}(e_1)\} \uplus \ldots \uplus \{\mathbb{T}(e_n)\}$$

(e) `FORALL` **quantifier**:

$$\mathbb{T}(\texttt{FORALL}\ v\ \texttt{IN}\ e_1 : e_2) =$$
$$[\,\mathbb{T}(e_2) \mid v \leftarrow \mathbb{T}(e_1)\,]^{all}$$

(f) `EXISTS` **quantifier**:

$$\mathbb{T}(\texttt{EXISTS}\ v\ \texttt{IN}\ e_1 : e_2) =$$
$$[\,\mathbb{T}(e_2) \mid v \leftarrow \mathbb{T}(e_1)\,]^{some}$$

(g) **"False" aggregate** (aggregate over a column reference in a grouping, $agg \in \{\texttt{SUM}, \texttt{MAX}, \texttt{MIN}\}$):

$$\mathbb{T}(\texttt{COUNT}(a)) = [\, 1 \mid G \leftarrow group \,]^{sum}$$

$$\mathbb{T}(agg(a)) = [\, G.\mathbb{col}(1, a) \mid G \leftarrow group \,]^{\mathbb{agg}(agg)}$$

$\mathbb{agg}$ is the straightforward mapping from SQL aggregates to MCC monoids:

$$\mathbb{agg}(\texttt{SUM}) = sum \qquad \mathbb{agg}(\texttt{MAX}) = max \qquad \mathbb{agg}(\texttt{MIN}) = min \quad .$$

These cases of $\mathbb{T}$ refer to the identifier *group* which represents the current group of a partitioned relation. *group* is introduced by the translation rules for `GROUP BY`, see (j).

(h) **Aggregation**:

$$\mathbb{T}\left(\begin{array}{ll}\texttt{SELECT} & agg_1(a_1), \ldots, agg_n(a_n) \\ \texttt{FROM} & e_1, \ldots, e_m \\ \texttt{WHERE} & q \end{array}\right) =$$

$$\begin{array}{l}[\,(\mathbb{col}(1, agg_1(a_1)) : \mathbb{T}(agg_1(a_1)), \\ \quad \ldots, \\ \quad \mathbb{col}(n, agg_n(a_n)) : \mathbb{T}(agg_n(a_n))) \mid \\ \quad group \equiv [\,(\mathbb{col}(1, a_1) : \mathbb{T}(a_1), \ldots, \mathbb{col}(n, a_n) : \mathbb{T}(a_n)) \mid \\ \qquad \mathbb{rv}(1, e_1) \leftarrow \mathbb{T}(e_1), \ldots, \mathbb{rv}(m, e_m) \leftarrow \mathbb{T}(e_m), \mathbb{T}(q)\,]^{bag}\,]^{bag}\end{array}$$

(i) `GROUP BY`:

$$\mathbb{T}\left(\begin{array}{ll}\texttt{SELECT} & p_1, \ldots, p_n, agg_1(a_1), \ldots, agg_k(a_k) \\ \texttt{FROM} & e_1, \ldots, e_m \\ \texttt{GROUP BY} & g_1, \ldots, g_r \end{array}\right) =$$

$$\mathbb{T}\left(\begin{array}{ll}\texttt{SELECT} & p_1, \ldots, p_n, agg_1(a_1), \ldots, agg_k(a_k) \\ \texttt{FROM} & e_1, \ldots, e_m \\ \texttt{WHERE} & \texttt{TRUE} \\ \texttt{GROUP BY} & g_1, \ldots, g_r \\ \texttt{HAVING} & \texttt{TRUE} \end{array}\right)$$

(j) `GROUP BY` **constrained by** `HAVING`:

$$\mathbb{T}\left(\begin{array}{rl} \texttt{SELECT} & p_1,\ldots,p_n,agg_1(a_1),\ldots,agg_k(a_k) \\ \texttt{FROM} & e_1,\ldots,e_m \\ \texttt{WHERE} & q_1 \\ \texttt{GROUP BY} & g_1,\ldots,g_r \\ \texttt{HAVING} & q_2 \end{array}\right) =$$

$$\begin{array}{l}
[\,(\mathbb{col}(1,p_1) : \mathbb{T}(p_1),\ldots, \\
\quad \mathbb{col}(n,p_n) : \mathbb{T}(p_n), \\
\quad \mathbb{col}(n+1,agg_1(a_1)) : \mathbb{T}(agg_1(a_1)),\ldots, \\
\quad \mathbb{col}(n+k,agg_k(a_k)) : \mathbb{T}(agg_k(a_k))) \mid \\
\qquad P \leftarrow [\,(\mathbb{col}(1,p_1) : \mathbb{T}(p_1),\ldots, \\
\qquad\quad \mathbb{col}(1,p_n) : \mathbb{T}(p_n), \\
\qquad\quad group \quad : [\,(\mathbb{col}(1,a_1) : \mathbb{T}(a_1),\ldots,\mathbb{col}(k,a_k) : \mathbb{T}(a_k)) \mid \\
\qquad\qquad\qquad \mathbb{rv}(1,e_1) \leftarrow \mathbb{T}(e_1),\ldots,\mathbb{rv}(m,e_m) \leftarrow \mathbb{T}(e_m), \mathbb{T}(q_1), \\
\qquad\qquad\qquad G.\mathbb{col}(1,g_1) = \mathbb{T}(g_1),\ldots,G.\mathbb{col}(r,g_r) = \mathbb{T}(g_r)\,]^{bag}) \mid \\
\qquad\quad G \leftarrow [\,(\mathbb{col}(1,g_1) : \mathbb{T}(g_1),\ldots, \\
\qquad\qquad\quad \mathbb{col}(r,g_r) : \mathbb{T}(g_r)) \mid \\
\qquad\qquad\quad \mathbb{rv}(1,e_1) \leftarrow \mathbb{T}(e_1),\ldots,\mathbb{rv}(m,e_m) \leftarrow \mathbb{T}(e_m)\,]^{set}\,]^{bag}, \\
\quad \mathbb{T}(q_2)\,]^{bag}
\end{array}$$

(k) `ORDER BY`:

$$\mathbb{T}(e\ \texttt{ORDER BY}\ a_1, \ldots, a_n) \quad = \quad \mathit{sort}\ \big(\circledcirc\mathrm{rd}(a_1), \ldots, \circledcirc\mathrm{rd}(a_n)\big)\ \mathbb{T}(e)$$

with

$$\circledcirc\mathrm{rd}(a\ \texttt{ASC}) = (\mathbb{T}(a), \leq) \qquad \circledcirc\mathrm{rd}(a\ \texttt{DESC}) = (\mathbb{T}(a), \geq) \quad .$$

Let *sort* denote a function that can sort its input according to the criteria given in $(\ldots)$, producing an output of type $\mathit{list}\ \alpha$ if $\mathbb{T}(e) :: \mathit{bag}\ \alpha$ or $\mathbb{T}(e) :: \mathit{set}\ \alpha$.

(l) `UNION`, `EXCEPT`, `INTERSECT`:

$$\begin{aligned} \mathbb{T}(e_1\ \texttt{UNION}\ e_2) &= \mathbb{T}(e_1)\ \uplus\ \mathbb{T}(e_2) \\ \mathbb{T}(e_1\ \texttt{EXCEPT}\ e_2) &= \mathbb{T}(e_1)\ \setminus^{+}\ \mathbb{T}(e_2) \\ \mathbb{T}(e_1\ \texttt{INTERSECT}\ e_2) &= \mathbb{T}(e_1)\ \cap\!\!\!+\ \mathbb{T}(e_2) \end{aligned}$$

$\uplus, \setminus^{+}, \cap\!\!\!+$ denote the respective bag operators.

(m) **Binary/unary operators** ($op \in \{\texttt{+}, \texttt{*}, \texttt{AND}, \texttt{OR}, \texttt{=}, \texttt{<>}, \texttt{<=}, \texttt{>=}, \texttt{<}, \texttt{>}, \texttt{NOT}\}$):

$$\begin{aligned} \mathbb{T}(e_1\ op\ e_2) &= \mathbb{T}(e_1)\ \circledcirc\mathbb{p}(op)\ \mathbb{T}(e_2) \\ \mathbb{T}(op\ e) &= \circledcirc\mathbb{p}(op)\ \mathbb{T}(e) \end{aligned}$$

with

$$\begin{array}{llll} \circledcirc\mathbb{p}(\texttt{+}) = + & \circledcirc\mathbb{p}(\texttt{*}) = * & \circledcirc\mathbb{p}(\texttt{AND}) = \wedge & \circledcirc\mathbb{p}(\texttt{OR}) = \vee \\ \circledcirc\mathbb{p}(\texttt{=}) = = & \circledcirc\mathbb{p}(\texttt{<>}) = \neq & \circledcirc\mathbb{p}(\texttt{<=}) = \leq & \circledcirc\mathbb{p}(\texttt{>=}) = \geq \\ \circledcirc\mathbb{p}(\texttt{<}) = < & \circledcirc\mathbb{p}(\texttt{>}) = > & \circledcirc\mathbb{p}(\texttt{NOT}) = \neg\ . & \end{array}$$

(n) **Parenthesized queries, ignoring tuple variables**:

$$\begin{aligned} \mathbb{T}(\ (e)\ ) &= \mathbb{T}(e) \\ \mathbb{T}(e\ \texttt{AS}\ v) &= \mathbb{T}(e) \end{aligned}$$

(o) **Basic query elements** (constants, column references, ...):

$$\mathbb{T}(e) = e$$

2.4.1 Applying $\mathbb{T}$ to Translate SQL Queries

- The definition of $\mathbb{T}$ is **compositional** in the sense that

  ① we can apply $\mathbb{T}$ to a complex SQL query q,
  ② then use $\mathbb{T}$ to translate the **subqueries** of q,
  ③ and finally assemble (compose) the resulting MCC parts to form the overall translation result (an MCC expression).

- This simplifies the implmentation of $\mathbb{T}$ and also helps us to easily track the progress of $\mathbb{T}$. Consider:

$$\mathbb{T}\left(\begin{array}{ll} \texttt{SELECT} & \texttt{COUNT}(*) \\ \texttt{FROM} & R\ \texttt{AS}\ x \\ \texttt{WHERE} & \texttt{EXISTS}\ y\ \texttt{IN}\ S : x.A = y.B \end{array}\right)$$

$$
\begin{array}{ll}
 & [\,(\mathtt{col}(1, \mathtt{COUNT}(*)) : \mathbb{T}(\mathtt{COUNT}(*)))\mid \\
=_{(h)} & \quad group \equiv \mathbb{T}\begin{pmatrix}\mathtt{SELECT} & * \\ \mathtt{FROM} & R\ \mathtt{AS}\ x \\ \mathtt{WHERE} & \mathtt{EXISTS}\ y\ \mathtt{IN}\ S : x.A = y.B\end{pmatrix}]^{bag} \\
 & [\,(X_1 : [\,1 \mid G \leftarrow group\,]^{sum})\mid \\
=_{(g)} & \quad group \equiv \mathbb{T}\begin{pmatrix}\mathtt{SELECT} & * \\ \mathtt{FROM} & R\ \mathtt{AS}\ x \\ \mathtt{WHERE} & \mathtt{EXISTS}\ y\ \mathtt{IN}\ S : x.A = y.B\end{pmatrix}]^{bag} \\
 & [\,(X_1 : [\,1 \mid G \leftarrow group\,]^{sum})\mid \\
=_{(b)} & \quad group \equiv [\,(\mathtt{col}(1, *) : \mathbb{T}(*))\mid \mathtt{rv}(1, R\ \mathtt{AS}\ x) \leftarrow \mathbb{T}(R\ \mathtt{AS}\ x), \\
 & \qquad\qquad \mathbb{T}(\mathtt{EXISTS}\ y\ \mathtt{IN}\ S : x.A = y.B), \\
 & \qquad\qquad * \equiv \mathtt{rv}(1, R\ \mathtt{AS}\ x).*\,]^{bag}\,]^{bag} \\
 & [\,(X_1 : [\,1 \mid G \leftarrow group\,]^{sum})\mid \\
=_{(f)} & \quad group \equiv [\,(\mathtt{col}(1, *) : \mathbb{T}(*))\mid x \leftarrow R, \\
 & \qquad\qquad [\,\mathbb{T}(x.A = y.B) \mid y \leftarrow \mathbb{T}(S)\,]^{some}, \\
 & \qquad\qquad * \equiv x.*\,]^{bag}\,]^{bag}
\end{array}
$$

$$
\begin{array}{ll}
\underset{(m)}{=} & [\,(X_1 : [\,1 \mid G \leftarrow group\,]^{sum}) \mid \\
& \qquad group \equiv [\,(\mathbb{co}\mathbb{l}(1, *) : \mathbb{T}(*)) \mid x \leftarrow R, \\
& \qquad\qquad [\,x.A = y.B \mid y \leftarrow \mathbb{T}(S)\,]^{some}, \\
& \qquad\qquad * \equiv x.*\,]^{bag}\,]^{bag} \\
\underset{(\equiv)}{=} & [\,(X_1 : [\,1 \mid G \leftarrow [\,(X_1 : x.*) \mid x \leftarrow R, \\
& \qquad\qquad [\,x.A = y.B \mid y \leftarrow S\,]^{some}\,]^{bag}\,]^{sum}) \mid\,]^{bag}
\end{array}
$$

- Note that, due to MCC semantics equation (1), this comprehension computes a result of type $(bag\ \mathbb{N})$ which is what we expect of the original SQL query.

✎ **Use $\mathbb{T}$ to compile this SQL query into an MCC expression (with $X :: bag\ (x : \mathbb{B}, y : \mathbb{N})$):**

$$\begin{array}{rl} \texttt{SELECT} & x, \texttt{SUM}(y) \\ \texttt{FROM} & X \\ \texttt{GROUP BY} & x \end{array}$$

- You will notice that you have to concern yourself with
  - **variable use and scoping** in monoid comprehensions,
  - comprehension semantics (see slide 80),
  - and **typing of subexpressions**.

2.5 Recommended Reading

[1] L. Fegaras and D. Maier, *Towards an Effective Calculus for Object Query Languages*, in Proc. of the ACM SIGMOD Int'l Conference on Management of Data, May 1995.

[2] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl, *Query Evaluation in CROQUE—Calculus and Algebra Coincide*, in Proc. of the 15th British National Conference on Databases (BNCOD15), C. Small, P. Douglas, R. Johnson, P. King, and N. Martin, eds., no. 1271 in Lecture Notes in Computer Science (LNCS), London, Birkbeck College, July 1997, Springer Verlag, pp. 84–100.

Contents