

- ▶ Relational database management systems (RDBMSs) provide the best understood and most carefully engineered query processing infrastructure (e.g., IBM DB2, Oracle, Microsoft SQL Server, PostgreSQL)

- ▶ Relational database management systems (RDBMSs) provide the best understood and most carefully engineered query processing infrastructure (e.g., IBM DB2, Oracle, Microsoft SQL Server, PostgreSQL)
- ▶ **Problem:**
 - ▶ RDBMSs are often operated as plain stores
 - ▶ Requires mastering of advanced features of query languages (e.g, SQL)
 - ▶ Pain to integrate into general purpose programming languages

- ▶ Relational database management systems (RDBMSs) provide the best understood and most carefully engineered query processing infrastructure (e.g., IBM DB2, Oracle, Microsoft SQL Server, PostgreSQL)
- ▶ **Problem:**
 - ▶ RDBMSs are often operated as plain stores
 - ▶ Requires mastering of advanced features of query languages (e.g, SQL)
 - ▶ Pain to integrate into general purpose programming languages
- ▶ **Solution:**
 - ▶ Use RDBMSs as a **co-processor** for general purpose programming languages, for those program parts that carry out **data-intensive** and **data-parallel** computations.

- ▶ Haskell has inspired a number of language integrated approaches:
 - ▶ LINQ (Meijer et al.)
 - ▶ Links (Wadler et al.)

- ▶ Haskell has inspired a number of language integrated approaches:
 - ▶ LINQ (Meijer et al.)
 - ▶ Links (Wadler et al.)
- ▶ Database-supported program execution in Haskell

Ferry Types

data Q a

Q $[(Int, String)]$

Q $[[Int], [String]]$

Ferry Combinators

<i>map</i>	$:: (QA\ a, QA\ b) \Rightarrow (Q\ a \rightarrow Q\ b) \rightarrow Q\ [a] \rightarrow Q\ [b]$
<i>append</i>	$:: QA\ a \Rightarrow Q\ [a] \rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>filter</i>	$:: QA\ a \Rightarrow (Q\ a \rightarrow Q\ Bool) \rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>head</i>	$:: QA\ a \Rightarrow Q\ [a] \rightarrow Q\ a$
<i>tail</i>	$:: QA\ a \Rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>init</i>	$:: QA\ a \Rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>length</i>	$:: QA\ a \Rightarrow Q\ [a] \rightarrow Q\ Int$
<i>reverse</i>	$:: QA\ a \Rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>concat</i>	$:: QA\ a \Rightarrow Q\ [[a]] \rightarrow Q\ [a]$
<i>take</i>	$:: QA\ a \Rightarrow Q\ Int \rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>drop</i>	$:: QA\ a \Rightarrow Q\ Int \rightarrow Q\ [a] \rightarrow Q\ [a]$
<i>zip</i>	$:: (QA\ a, QA\ b) \Rightarrow Q\ [a] \rightarrow Q\ [b] \rightarrow Q\ [(a, b)]$
<i>unzip</i>	$:: (QA\ a, QA\ b) \Rightarrow Q\ [(a, b)] \rightarrow Q\ ([a], [b])$
<i>groupWith</i>	$:: (Ord\ b, QA\ a, QA\ b) \Rightarrow (Q\ a \rightarrow Q\ b) \rightarrow Q\ [a] \rightarrow Q$
<i>sortWith</i>	$:: (Ord\ b, QA\ a, QA\ b) \Rightarrow (Q\ a \rightarrow Q\ b) \rightarrow Q\ [a] \rightarrow Q$
<i>the</i>	$:: (Eq\ a, QA\ a) \Rightarrow Q\ [a] \rightarrow Q\ a$
<i>table</i>	$:: TA\ a \Rightarrow String \rightarrow Q\ a$

More on Types

```
class QA a  
instance QA ()  
instance QA Bool  
instance QA Char  
instance QA Int  
instance QA a  $\Rightarrow$  QA [a]  
instance (QA a, QA b)  $\Rightarrow$  QA (a, b)  
...
```

Query Comprehensions

$$\text{dotp } sv \ v = \text{sum } [\$qc \mid x * (v ! i) \mid (i, x) \leftarrow sv \mid]$$

Example Data

facilities	
fac	cat
SQL	QLA
HDB	LIB
ODBC	API
LINQ	LIN
Links	LIN
Rails	ORM
Ferry	LIB
ADO.NET	ORM
Kleisli	QLA

features	
fac	feature
SQL	aval
SQL	type
SQL	SQL!
HDB	comp
HDB	type
HDB	SQL!
LINQ	nest
LINQ	comp
LINQ	type
Links	comp
Links	type
Links	SQL!
Rails	nest
Rails	maps
Ferry	list
Ferry	nest
Ferry	comp
Ferry	aval
Ferry	type
Ferry	SQL!
ADO.NET	maps
ADO.NET	comp
ADO.NET	type
Kleisli	list
Kleisli	nest
Kleisli	comp
Kleisli	type

meanings	
feature	meaning
list	respects list order
nest	supports data nesting
aval	avoids query avalanches
type	is statically type-checked
SQL!	guarantees translation to SQL
maps	admits user-defined object mappings
comp	has compositional syntax and semantics

Example Queries

hasFeatures :: *Q String* → *Q [String]*

hasFeatures f =

[*\$qc* | *feature* | (*fac, feature*) ← *features*, *fac* ≡ *f* |]

means :: *Q String* → *Q String*

means f =

head [*\$qc* | *meaning* | (*feature, meaning*) ← *meanings*, *feature* ≡ *f* |]

query :: *Q [(String, [String])]*

query =

[*\$qc* | (*the cat*, nub (concatMap (map means ∘ hasFeatures) *fac*))
| (*fac, cat*) ← *facilities*
, **then** group by *cat*]

```
[("API", []),  
 ("LIN", ["avoids query avalanches",  
          "guarantees translation to SQL",  
          "has compositional syntax and semantics",  
          "is statically type-checked",  
          "respects list order",  
          "supports data nesting"]),  
 ("LIB", ["guarantees translation to SQL",  
          "has compositional syntax and semantics",  
          "is statically type-checked"]),  
 ("ORM", ["admits user-defined object mappings",  
          "is statically type-checked",  
          "supports data nesting"]),  
 ("QLA", ["avoids query avalanches",  
          "guarantees translation to SQL",  
          "has compositional syntax and semantics",  
          "is statically type-checked",  
          "respects list order",  
          "supports data nesting"])]
```

More on Classes

class $QA\ a$ **where**

$toQ :: a \rightarrow Q\ a$

$fromQ :: Conn \rightarrow Q\ a \rightarrow IO\ a$

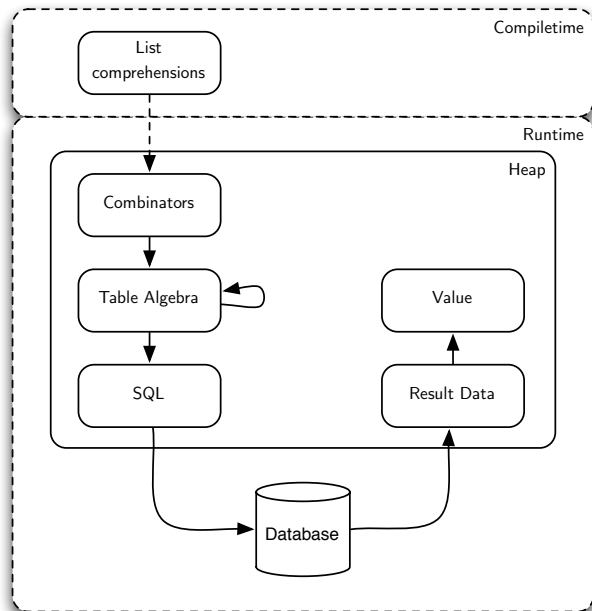
class $QA\ a \Rightarrow TA\ a$ **where**

$table :: String \rightarrow Q\ [a]$

class $View\ a\ b \mid a \rightarrow b$ **where**

$view :: a \rightarrow b$

Code Motion



Phantom Types

```
data Exp =  
  VarE String  
  | UnitE  
  | BoolE Bool  
  | CharE Char  
  | IntE Int  
  | TupleE Exp Exp [Exp]  
  | ListE [Exp]  
  | FuncE (Exp → Exp)  
  | AppE Exp Exp  
  | TableE String Type
```

```
data Q a = Q Exp
```

```
take :: (QA a) ⇒ Q Int → Q [a] → Q [a]
```

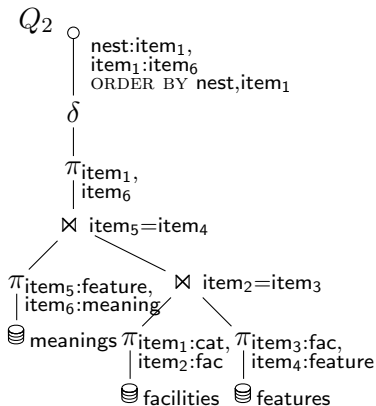
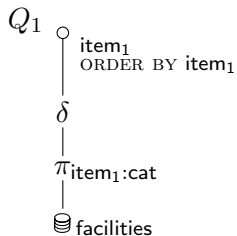
```
take (Q i) (Q as) = Q (AppE (AppE (VarE "take") i) as)
```

Table Algebra

Operator	Semantics						
$\pi_{a_1:b_1, \dots, a_n:b_n}$	project onto columns b_i , rename b_i into a_i						
σ_p	select rows satisfying predicate p						
$- \times -$	Cartesian product						
$- \bowtie_p -$	join with predicate p						
δ	eliminate duplicate rows						
$- \dot{\cup} -$	disjoint union						
$- \setminus -$	difference						
$@_{a:v}$	attach constant value v in column a						
$\text{CAST}_{a:(t)}b$	attach value of b casted to type t in a						
$\otimes_{a:\langle b_1, \dots, b_n \rangle}$	attach result of application $*$ (b_1, \dots, b_n) in a						
$\varrho_{a:\langle b_1, \dots, b_n \rangle/c}$	group by c , attach row rank (in b_i order) in a						
$\text{AGG}_{a:\langle b \rangle/c}$	group by c , compute aggregate of b in a						
\textcircled{R}	read from database-resident table R						
<table border="1"><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>-</td><td>-</td><td>-</td></tr></table>	a	b	c	-	-	-	literal table with columns a, b, c
a	b	c					
-	-	-					

Table: Intermediate table algebra (with n -ary operator $* \in \{+, =, \text{and}, \dots\}$ and $\text{AGG} \in \{\text{COUNT}, \text{MAX}, \text{MIN}, \dots\}$).

Example Query Plan Bundle



Example SQL Bundle

```
SELECT DISTINCT t0000.cat AS item1
  FROM facilities AS t0000
ORDER BY t0000.cat ASC;
```

```
SELECT DISTINCT t0001.cat AS nest, t0000.meaning AS item1
  FROM meanings AS t0000,
       facilities AS t0001,
       features AS t0002
 WHERE t0000.feature = t0002.feature
       AND t0001.fac = t0002.fac
ORDER BY t0001.cat, t0000.meaning ASC;
```

Nesting

	positem ₁	...	item _n
1	v_{11}	...	v_{1n}
2	v_{21}	...	v_{2n}
\vdots	\vdots		
ℓ	$v_{\ell 1}$...	$v_{\ell n}$

(a) Encoding a flat ordered list.

Q_0		Q_1			
pos	item ₁	nest	positem ₁	...	item _n
1	@ ₁	@ ₁	1		x_{11}
\vdots	\vdots	\vdots	\vdots		\vdots
n	@ _n	@ ₁	ℓ_1		$x_{1\ell_1}$
		\vdots	\vdots		\vdots
		@ _n	1		x_{n1}
		\vdots	\vdots		\vdots
		@ _n	ℓ_n		$x_{n\ell_n}$

(b) Encoding a nested list (Q_0 : outer list, Q_1 : all inner lists).

Figure: Relational runtime encoding of order and nesting on the database back-end.

Related Work

- ▶ LINQ and Links
- ▶ Data-parallel Haskell
- ▶ Iteratively Staged Embedding

Questions/Comments/Suggestions