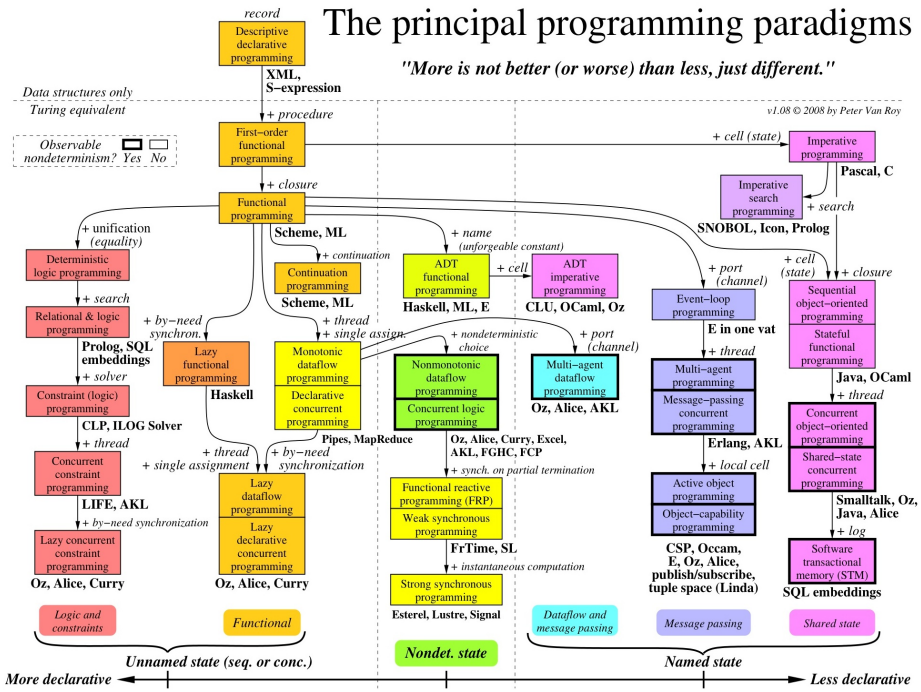
TDDA69 Data and Program Structure
Relational Programming

Cyrille Berger

LINKÖPING UNIVERSITY



Lectures

Lecture content

- Relational Programming
- SQL And Databases
  - Interpreting SQL
- Logic Programming

Relational Programming

Relational Programming

- A relational programming language is a declarative language built around the relational model of data:
  - Defines
  - Defines relations between
- Examples: **SQL**

Relational vs Functional

- In Functional, outputs are functions of the inputs

  f(1,2)
- In Relational, arguments can be inputs or outputs

  f(?answer, 1) f(2, ?question) f(2, 3) f(2, 4)...

  ?answer = ∅ ?question = 3 or 4

Relational Syntax

- Syntax, introduce *let* and *ask*

```
let soft(beige); let soft(coral)
let hard(mauve); let hard(ochre);
let contrast(soft, hard);
let contrast(hard, soft);
value color;
ask contrast(beige, color);
```

SQL And Databases

Database Management Systems

- A Database is a collection of table
- A table is a collection of records
- A record is a row with a value for each column
- A column has a name and a type

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

- The Structured Query Language (SQL) is the most widely used programming language for accessing DBMS

Select Statements Project Existing Tables

- Select statements:
  `SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order]`
- A **select** statement can specify an input table using a **from** clause
  `SELECT parent, child FROM parents;`
- A subset of the rows of the input table can be selected using a **where** clause
  `SELECT parent, child FROM parents WHERE parent='fillmore';`
- An ordering over the remaining rows can be declared using an **order by** clause
  `SELECT parent, child FROM parents ORDER BY parent`
- Column descriptions determine how each input row is projected to a result row
  `SELECT child FROM parents ORDER BY parent`

SQL Examples

- Table creation

```
create table cities as
  select 38 as latitude, 122 as longitude, "Berkeley" as name union
  select 42,             71,               "Cambridge"
union
  select 45,             93,
"Minneapolis";
```

Latitude	Longitude	Name	Region	Name
38	122	Berkeley	west coast	Berkeley
42	71	Cambridge	other	Minneapolis
45	93	Minneapolis	other	Cambridge

- Data retrieval

```
select "west coast" as region, name from cities where longitude >=
115 union
select "other", name from cities where longitude < 115;
```

Joining Multiple Tables

- Multiple tables can be joined to yield all combinations of rows from each

```
create table grandparents as
select a.parent as grandog, b.child as granpup
from parents as a, parents as b
where b.parent = a.child;
```

- Select all grandparents with the same fur as their grandchildren

```
select grandog from grandparents, dogs as c, dogs as
d
where grandog = c.name and
      granpup = d.name and
      c.fur = d.fur;
```


Interpreting SQL

Table representation in Python

- The namedtuple function returns a new sub-class of tuple

```
from collections import namedtuple
City = namedtuple('City', ['latitude', 'longitude', 'name'])
cities = [City(38, 122, 'Berkeley'),
City(42, 71, 'Cambridge'),
City(43, 93, 'Minneapolis')]
[city.latitude for city in cities]
```

-> [38, 42, 43]

- Attribute names are accessible as the _fields attribute of an instance of City

```
print(cities[0])
print(cities[0]._fields)
```

Output:

```
-> City(latitude=38, longitude=122, name='Berkeley')
-> ('latitude', 'longitude', 'name')
```


A naive first implementation of select

- One correct (but not always efficient) implementation of select uses sequence operations
- Example of query:

```
SELECT name, 60*abs(latitude-38) AS distance FROM cities WHERE name !
= 'Berkeley';
Distance = namedtuple('Row', ['name', 'distance'])
def columns(city):
    latitude, longitude, name = city
    return Distance(name, 60*abs(latitude-38))
def condition(city):
    latitude, longitude, name = city
    return name != 'Berkeley'
for row in map(columns, filter(condition, cities)):
  print(row)
-> Row(name='Cambridge', distance=240)
-> Row(name='Minneapolis', distance=300)
```

SQL Interpretation Example

- SQL Queries:

```
CREATE TABLE cities AS
SELECT 38 AS lat, 122 AS lon, 'Berkeley' AS name UNION
SELECT 42, 71, 'Cambridge' UNION
SELECT 45, 93, 'Minneapolis';
SELECT 60*(lat-38) AS north FROM cities WHERE name != 'Berkeley';
```

- In Python:

```
City = namedtuple('City', ['lat', 'lon', 'name'])
cities = [City(38, 122, 'Berkeley'),
          City(42, 71, 'Cambridge'),
          City(43, 93, 'Minneapolis')]
s = Select('60*(lat-38) as north', 'cities', 'name != "Berkeley"')
for row in s.execute({'cities': cities}):
    print(row)
```


A Select Class

- The SQL parser creates an instance of the Select class for each select statement

```
class Select:
    """select [columns] from [tables] where [condition]."""
    def __init__(self, columns, tables, condition):
        self.columns = columns
        self.tables = tables
        self.condition = condition
        self.make_row = create_make_row(self.columns)
    def execute(self, env):
        """Join, filter, and map rows from tables to columns."""
        from_rows = join(self.tables, env)
        filtered_rows = filter(self.filter_fn, from_rows)
        return map(self.make_row, filtered_rows)
    def filter_fn(self, row):
        if self.condition:
            return eval(self.condition, row)
        else:
            return True
```


Creating Row Classes Dynamically

- Each select statement creates a table with new columns, represented by a new class

```
def create_make_row(description):
    """
    Return a function from an input environment (dict) to an output row.
    description -- a comma-separated list of [expression] as [column name]
    """
    columns = description.split(', ')
    expressions, names = [], []
    for column in columns:
        if ' as ' in column:
            expression, name = column.split(' as ')
        else:
            expression, name = column, column
        expressions.append(expression)
        names.append(name)
    row = namedtuple('Row', names)
    return lambda env: row(*[eval(e, env) for e in expressions])
```


Joining Rows

- Joining creates a dictionary with all names and aliases for each combination of rows

```
from itertools import product
def join(tables, env):
    """Return an iterator over dictionaries from names to values in a
row."""
    names = tables.split(', ')
    joined_rows = product(*[env[name] for name in names])
    return map(lambda rows: make_env(rows, names), joined_rows)
def make_env(rows, names):
    """Create an environment of names bound to values."""
    env = dict(zip(names, rows))
    for row in rows:
        for name in row._fields:
            env[name] = getattr(row, name)
    return env
```

Query Planning

- The manner in which tables are filtered, sorted, and joined affects execution time
- Select the parents of curly-furred dogs:

```
select parent from parents, dogs
    where child = name and fur = 'curly';
```

- Four different possibilities:
  - Join all rows of parents to all rows of dogs, filter by child = name and fur = 'curly'
  - Join only rows of parents and dogs where child = name, filter by fur = 'curly'
  - Filter dogs by fur = 'curly', join result with all rows of parents, filter by child = name
  - Filter dogs by fur = 'curly', join only rows of result and parents where child = name

Logic Programming

Logic Programming

- Based on Formal logic: expressing facts and rules
- Examples: Prolog

Predicate logic

- Predicate logic can be used to capture facts and rules:
  - declare facts as ground clauses.
    E.g., Son(Gustaf, Carl), Daughter (Carl, Victoria), ...
  - rules as horn clauses:
    $\forall$ x,y, z Son(x, y) ^ Daughter (y, z) $\supset$ GrandFather (x, z)
  - One can then submit queries and retrieve further facts:
    $\exists$ x GrandFather (Gustaf, x)

Query answering system

- Knowledge is stored in a database and is represented:
  - explicitly as facts
  - or implicitly as rules
- An inference machine infers new facts from known ones
- Programs submit queries
  - A query is simple or composed of simple queries and the connectives and, or, not
  - Queries are compared against the knowledge in the database by **pattern matching** for the facts and by **unification** for the rules

Pattern matching and unification

- **Pattern matching**
  - Match a query with variables to facts without variables

```
(query (parent abraham ?child))
(fact (parent abraham barack))
```

- In **Unification**:
  - Unification is a generalization of pattern matching.

```
(query (grandparent abraham ?grandchild)
(fact (parent barack lincoln))
(rule (grandparent ?x ?z) (parent ?x ?y) (parent ?y ?z)
```

  - Unification finds bindings for variables.
  - A variable occurring several times will be bound to the same value
  - In unification, a variable can be bound to another expression or variable

Unification algorithm

- Unification is a generalization of pattern matching that attempts to find a mapping between two expressions that may both contain variables.
- Example:

  (?x ?x) can match ((a ?y c) (a b ?z))
  ((a b c) (a b c))
- Unification identifies this solution via the following steps:
  - To match the first element of each pattern, the variable ?x is bound to the expression (a ?y c).
  - To match the second element of each pattern, first the variable ?x is replaced by its value. Then, (a ?y c) is matched to (a b ?z) by binding ?y to b and ?z to c.

Unification Algorithm

unify(e, f,

1) Both inputs *e* and *f* are replaced by their values if they are variables.
2) If *e* and *f* are equal, unification succeeds.
3) If *e* is a variable, unification succeeds and *e* is bound to *f*.
4) If *f* is a variable, unification succeeds and *f* is bound to *e*.
5) If neither is a variable, both are not lists, and they are not equal, then e and f cannot be unified, and so unification fails.
6) If none of these cases holds, then *e* and *f* are both pairs, and so unification is performed on both their first and second corresponding elements.

Unification Algorithm - Example

- unify((?x ?x), ((a ?y c) (a b ?z)))
- unify(?x, (a ?y c))
  ?x = (a ?y c)
- unify(?x, (a b ?z))
  unify((a ?y c), (a b ?z))
- unifiy(a, a)
- unifiy(?y, b)
  ?y = b
- unifiy(c, ?z)
  ?z = c

Query interpreter

- The query interpreter performs a search in the space of all possible facts
- Unification is the primitive operation that pattern matches two expressions
- It is a recursive algorithm

Search Algorithm

- The process of attempting to demonstrate an assertion (answer a query) is a systematic depth-first search of facts.

```
def search(clauses, env):
    if clauses is nil:
        yield env
    for fact in facts:
        fact = rename_variables(fact, get_unique_id())
        env_head = new environment that extends env
        if unify(fact.first, clauses.first, env_head):
            for env_rule in search(fact.second, env_head, depth+1):
                for result in search(clauses.second, env_rule,
depth+1):
                    yield result
```


Conclusion

- Relational Programming
- Logic programming
  - and how to infer new facts