Specifying Rule-based Query Optimizers in a Reflective Framework

Leonidas Fegaras
David Maier
Tim Sheard

Oregon Graduate Institute

This work

Part of the **EREQ** project:
Architectures for Query Processing in Persistent Object Bases

Contributions in three areas:

- a new declarative language for specifying optimizers;
- an attribute framework for propagating information during query transformation and planning;
- a reflective language environment for translating optimizer specifications into query optimizers.

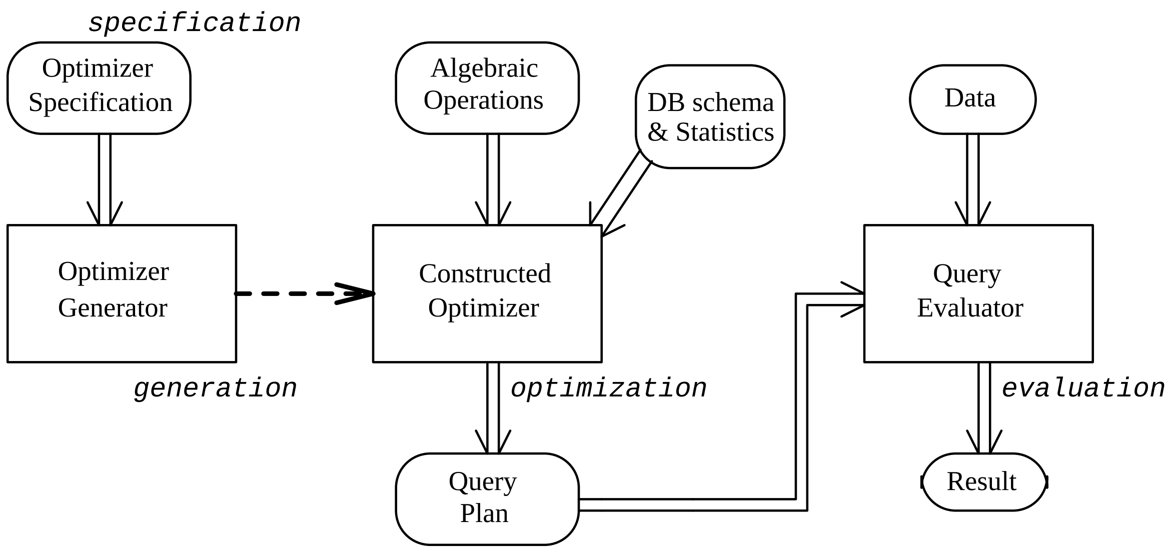
Problems with current query optimizers

Most current optimizer generator frameworks are not that declarative:

Their value is in **knowledge engineering**
(kinds of information and its partitioning).

We are concentrating on **knowledge representation** and **translation**.

Reference Architecture



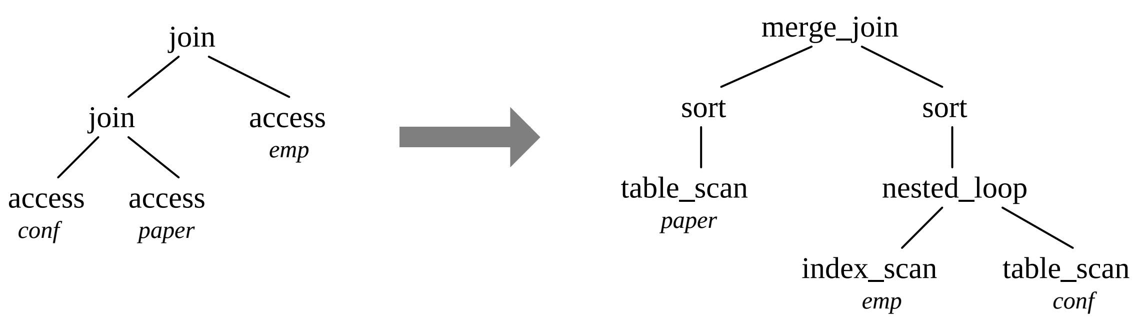
What needs to be specified

- logical datatypes and physical storage structures;
- signatures of logical operators;
- physical algorithms;
- logical transforms;
- implementation rules;
- logical properties;
- physical properties (desired and generated);
- cost functions;
- control strategy.

Representation of logical expressions and physical plans

Logical expression:

```
<< project( join( join( access(conf,[conf.name="VLDB"]),
                        access(paper,[paper.subject="DB"]),
                        [paper.year=conf.year]),
                  access(emp,[emp.status="Professor"]),
                  [ emp.eno=paper.eno,
                    emp.eno=conf.eno ]),
            [emp.name] ) >>
```



One possible QEP is:

```
<< Project(
     merge_join( sort( table_scan(paper,[paper.subject="DB"]),
                       [paper.eno,paper.year] ),
                 sort( nested_loop( index_scan(emp,[emp.status],
                                               [emp.status="Professor"]),
                                    table_scan(conf,[conf.name="VLDB"]),
                                    [emp.eno=conf.eno] ),
                       [emp.eno,conf.year] ),
                 [ emp.eno=paper.eno,
                   paper.year=conf.year ] ),
     [emp.name] ) >>
```

Rewrite rules

Rules take the form:

pattern = list of expressions

expressions can be given by free constructions of expression trees or by function result.

Implementation rule:

```
<< join( ‘x, ‘y, ‘p ) >>
    = [ << nested_loop(‘x,‘y,‘p) >> ]
```

Transformation rules:

```
<< join(‘x,‘y,‘p1 and ‘p2) >>
    = [ << intersect(join(‘x,‘y,‘p1),join(‘x,‘y,‘p2)) >> ]
```

```
<< map(fn ‘x => ‘e) (map(fn ‘y => ‘u) ‘z) >>
    = [ << map(fn ‘y => ‘(subst(e,x,u))) ‘z >> ]
```

which implements the algebraic transformation

$$\mathrm{map}(\lambda x.e(x))(\mathrm{map}(\lambda y.u(y))\,z) \;\rightarrow\; \mathrm{map}(\lambda y.e(u(y)))\,z$$

Attribute framework

Query expressions carry annotations at each node that are propagated during rewrites

Idea borrowed from **attribute grammars**

Types of attributes:

Inherited: propagated across rewrites
defined in each rewrite rule
e.g. the required sort order for a query plan

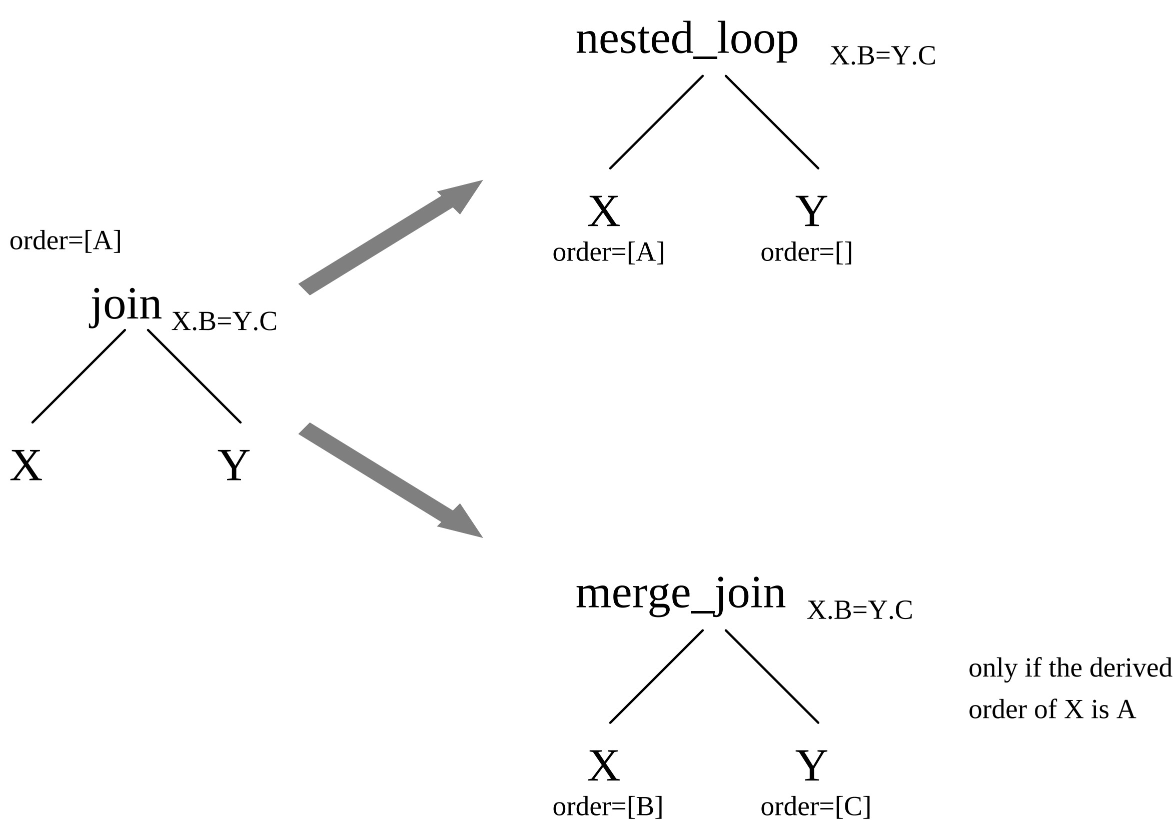
Synthesized: constructed up the expression tree
defined in one place
e.g. cardinality, cost

Our attribute framework can capture most kinds of information used in other optimizer frameworks:

- logical properties;
- physical properties (expected and generated);
- costs;
- context information;
- control strategies.

Rewrites with attributes

```
{order=exp_ord}
   << join( ‘x <= { order=exp_ord },
            ‘y <= { order=[] },
            ‘p ) >>
   = [ << nested_loop(‘x,‘y,‘p) >> ]

{order=exp_ord}
   << join( ‘x <= { order=required_order(tables(x),p) },
            ‘y <= { order=required_order(tables(y),p) },
            ‘p ) >>
      = if subsumes(exp_ord,#order x)
           then [ << merge_join(‘x,‘y,‘p) >> ]
           else []
```



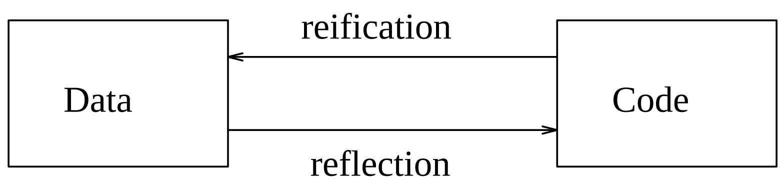
Rewrites with attributes (cont.)

```
<< access(‘t,‘p) >>
      = [ << table_scan(‘t,‘p) >> ]

{order=ord,indices=idx}
   << access(‘tbl,‘p) >>
      = (map(fn (s,c) => << index_scan(‘tbl,‘c,‘p) >>)
            (filter(fn (s,c) => (s=tbl)
                                    andalso (subsumes(ord,c))))
                   idx)

{order=(a::r)}
   << ‘x <= {order=[]} >>
      = if subsumes(a::r,#order x)
           then [ << sort(‘x,‘(a::r)) >> ]
           else []
```

Specification processing

Our specification language as well as the specification processing is expressed in **CRML**: Compile-time Reflective ML

What is *reflection*?

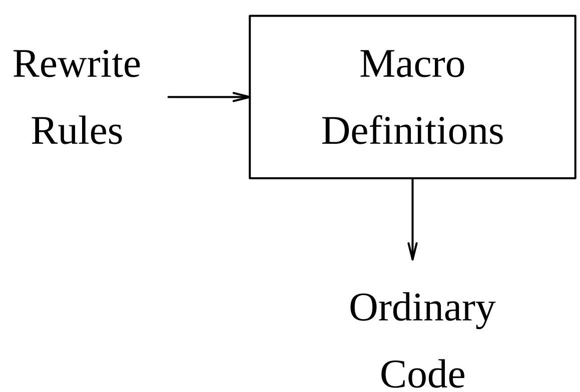


Reflection removes a layer of interpretation by generating customized programs.

Our specification language consists of a fixed set of CRML macros for:

- attribute declaration;
- computation of synthesized attributes;
- specification of rewrite rules.

Specifications are translated directly into SML code (functions):



Search strategy

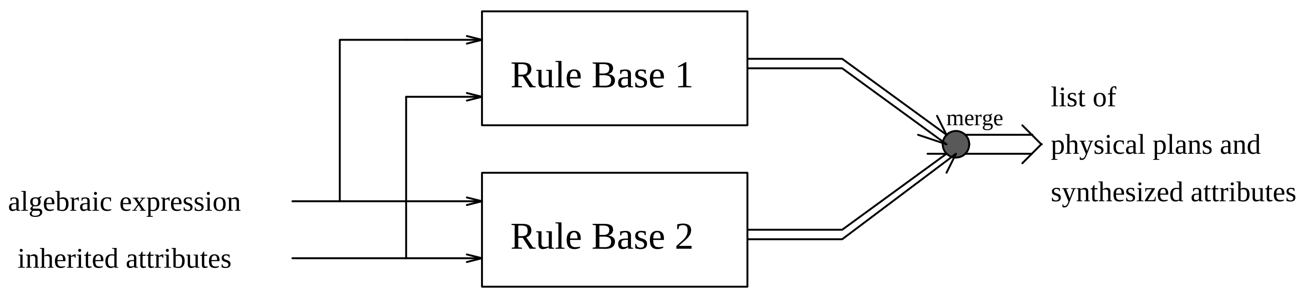
The search engine is generated from the rule-base specifications:
it is tailored individually to each different rule-base.

The search engine

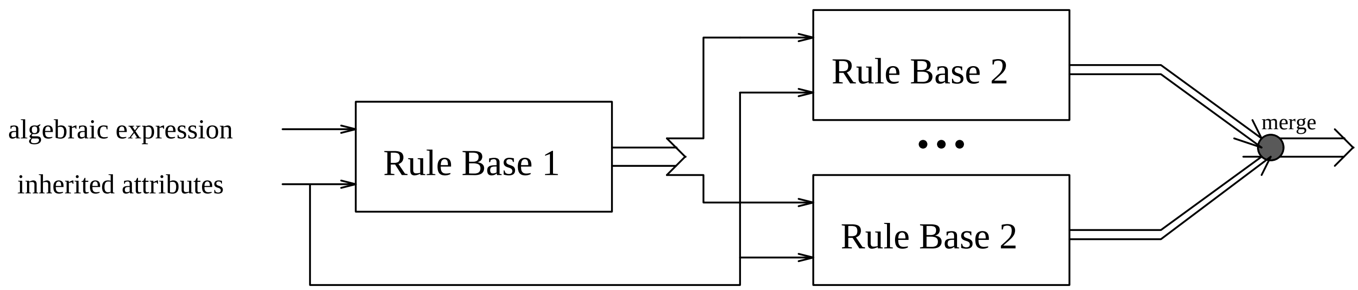
- is a recursive functions that calls itself to some carefully selected points;
- is a high-order function, parameterized by parts of the rewriting process;
- doesn't need any pattern matching during rule evaluation;
- supports memoization;
- considers all applicable rules each time;
- constructs expression trees bottom-up;
- accumulates all valid plans in a user-defined way.

Composition of Rule-Based Modules

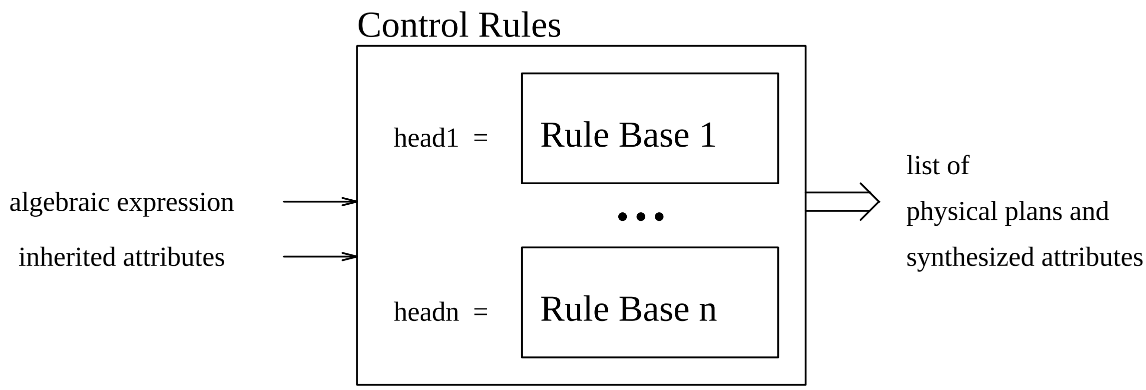
Vertical Composition:



Horizontal Composition:



Hierarchy of Rules:



Current Exercises

- an optimizer for system R;
- an optimizer for a query algebra based on structural recursion;
- an optimizer for the TI Open OODB system;
- retargeting to Volcano optimizer generator;
- support for heuristic guidance and pruning.