Safe Query Objects: Statically-Typed Objects as Remotely-Executable Queries

William Cook
Department of Computer Sciences
University of Texas at Austin
cook@cs.utexas.edu

Siddhartha Rai
Department of Computer Sciences
University of Texas at Austin
sid@cs.utexas.edu

ABSTRACT

When building scalable systems that involve general-purpose computation and persistent data, object-oriented languages and relational databases are often essential components. Yet the impedance mismatch between these technologies has not been completely overcome by existing integration approaches. Call level interfaces like ODBC and JDBC are an unsafe and fragile form of metaprogramming: database queries are constructed at runtime as strings and executed as programs against the database engine. Object/relational mapping and persistent object systems do not support query shipping, in which complex queries are sent to the database for execution. This paper presents *safe query objects*, an integrated approach to impedance mismatch that allows query behavior to be defined using statically-typed objects and methods. In addition, safe query objects support query shipping by automatically generating code to execute queries remotely in a relational database. A concrete implementation of this solution is presented using the OpenJava macro language and Java Data Objects.

1. INTRODUCTION

In 1984, David Maier introduced the term *impedance mismatch* to describe the difficulty of integrating programming languages and databases [16]. In this paper we focus on an important but specific instance of this problem: how to integrate *object-oriented programming languages* with *relational databases* [3]. In the last 20 years numerous approaches have been proposed to solve this problem, including call level interfaces [24, 10, 18], object/relational mapping [13, 6, 17], and persistent object systems [2, 21, 5]; each of these have been investigated at least a dozen times. Despite steady progress toward a solution, impedance mismatch is still an open problem.

The lack of overwhelming success for any of these solutions suggests a more fundamental question: What exactly is the problem? Maier offered an important hint: "Whatever the database programming model, it must allow complex, data-intensive operations to be picked out of programs for execution by the storage manager, rather than forcing a record-at-a-time interface." A close examination of the benefits and drawbacks of the range of solutions described above leads to two principles:

1. The entire program, including queries and other database operations, must be specified within a *unified semantic framework*. This does not require a single language for all aspects of a system, it only requires that there be no semantic disconnects between the different parts. The unifying framework should also enable a large degree of static type-checking across all parts of the program.

2. Operations must be *executed efficiently* by leveraging appropriate optimizations in both compilers and database engines. In practice this means that parts of the behavior of a program must be executed in the database. Unfortunately the behavior that the database *can* execute and the behavior that it *cannot* are sometimes tightly intertwined; as a result it is not acceptable for a proposed solution to enforce simplistic modularity between the two.[1]

The widely-used solutions to impedance mismatch are partial because they focus on one or the other of these goals, but not both. Persistent object systems and object/relational mapping define a unifying semantics of persistent data as objects, but do not provide a unified mechanism for leveraging the power of databases for optimized search algorithms, caching, and indexing. Call level interfaces provide direct access to all the power of a database engine, but introduce a semantic disconnect between programming language code and embedded database code.

Recently progress has been made at tighter semantic integration for call level interfaces [8]. Some object-based systems [6, 5, 21, 17] include a form of call level interface; however, the resulting hybrid is a combination of two partial solutions, not a unified solution. Query languages may also be embedded directly in a programming languages, as in SQLJ [1], but this requires language changes and also is limited to static queries. HaskellDB [15], achieves both goals simultaneously in the context of a pure functional language and a subset of SQL.

This analysis suggests that impedance mismatch is caused by the difficulty of simultaneously meeting both goals. Requirement #2 is particularly difficult because programming languages do not typically support breaking off large parts of a program and translating them for execution in a different environment. This is related to the general problem of mobile code [11], but has the added complication of interfacing to relational query languages.

[1] An example is given in Section 4.3.

This paper presents *safe query objects*, a new approach to overcoming impedance mismatch. Safe queries are Java objects that follow a specific programming pattern to define the behavioral components of a query. Since safe queries are Java objects, they are semantically integrated and typed-checked statically, and can be executed locally within the Java VM. Safe queries leverage database optimization by supporting *remote execution*, in which complex query behavior is sent to the database for execution. The methods for remote execution are generated automatically at compile time by translating the query behavior into equivalent code to invoke remote database operations through a call level interface.

The prototype of safe query objects uses OpenJava [23] for compile-time meta-programming and generates code to invoke the Java Data Objects (JDO) persistence library [21]. Although JDO is a practical target to use because of its industrial significance, the prototype is limited by the capabilities of JDO. JDO supports a significant subset of relational query behavior: filtering (including dynamic filters and filters that involve joins between multiple tables), sorting, parameterization, and existential quantification. It does not cover multi-table query results or general aggregation. The prototype of safe query objects presented here inherits these capabilities and limitations. While OpenJava and JDO are used in the prototype implementation, the solution is not specific to either. This paper is a first step towards a comprehensive solution to impedance mismatch that will support relational database functionality in its full generality. Future work will address update operations in addition to queries involving general joins and aggregation.

The remainder of this paper is organized as follows: Section 2 reviews background and motivates the problem. Section 3 presents an overview of safe query objects, using filtering as an example. Section 4 elaborates the basic model to handle sorting, parameterized queries, dynamic filters and quantification in filters. Section 5 provides details on the generation of code for remote execution of a query object. Section 6 evaluates the current results and discusses directions for future work. Section 7 reviews related work and Section 8 presents conclusions.

2. BACKGROUND AND MOTIVATION

2.1 Call Level Interfaces

A call level interface (CLI) allows a programming language to access a database engine through a standardized API [24, 10, 18]. The Java code in Figure 1 uses JDBC [10] to execute a SQL query and process the results. The sequence of calls begin a session (`getConnection`), create an execution context (`createStatement`), execute queries (`executeQuery`), iterate through resulting rows (`next`) and access result columns (`getString`) is typical of CLI, although there are many variations. For example, some call level interfaces use objects to represent the abstract syntax tree of a query [5, 4], rather than strings.

Call level interfaces are successful because they are the simplest way to achieve *query shipping* [7], which is often considered essential to the performance of practical systems built around relational databases. *Query shipping* refers to the practice of transferring high-level operations, like filtering, joining, sorting, and aggregation, to the database when this improves performance. The alternative strategy, *data shipping*, uses simpler queries to return data to the client for processing. In what follows, performing high-level operations by data shipping is called *local execution*, and query shipping is called *remote execution*.

```
float limit = 50000;
Connection con = DriverManager.getConnection(...);
Statement stmt = con.createStatement();
String sql = "SELECT * FROM Employee "
    + "WHERE salary > " + Float.toString(limit);
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
  String name = rs.getString("Name");
  float salary = rs.getFloat("Salary");
  print(name + ": " + Float.toString(salary));
}
```

Figure 1: Query execution with JDBC

Despite their usefulness, call level interfaces have a number of significant problems. First, the embedded database programs are not checked until they are passed to the CLI at runtime. As a result, the syntax and types of database programs are not checked statically, but instead result in runtime errors. This is true despite the fact that the database structure is almost always static and known when the client program is compiled.

Second, programs that use call level interfaces are difficult to write and maintain. There are important classes of queries that must be constructed dynamically at runtime. Manipulating programs as strings is complex and error-prone due to the complex rules for nesting expressions, quoting constants, and the interplay between embedded and host languages. Concepts that are relatively straightforward to express in a language, like query parameters, are awkward to specify and invoke via an API. Query results are represented as untyped objects that are accessed by string names. There are many subtle but unchecked dependencies between the query being executed and the code that decodes its results.

Finally, call level interfaces make it difficult to reuse queries; doing so involves complex manipulation of programs at runtime, while avoiding name conflicts and ensuring consistency of the resulting query.

These problems lead to fragile systems that are difficult to build and maintain. A tremendous amount of effort is expended in industry every year in trying to solve these problem and deal with the limitations of existing solutions.

2.2 Metaprogramming and OpenJava

A *metaprogram* is a program whose input or output is another program. There are many forms of metaprogramming, depending on whether programs are being analyzed or generated (or both), whether the process happens at runtime or compile-time, and the degree of type checking performed [22].

Call level interfaces are a form of runtime metaprogram-

```
class A instantiates M {
   int a;
}
```

(a) *Sample program using metaclass M*

```
class M extends OJClass {
  public void translateDefinition() {
    OJField[] fields = getFields();
    for (int i=0; i< fields.length; i++) {
      addField(new OJField(this,
        OJModifier.forModifier(OJModifier.PUBLIC),
        OJClass.forName("java.lang.String"),
        fields[i].getName() + "_str"));
    }
  }
}
```

(b) *Definition of metaclass M*

```
class A {
   int    a;
   String a_str;
}
```

(c) *Java Class generated by OpenJava*

Figure 2: OpenJava Example

ming. The Java code in Figure 1 is constructing and executing a database program, in this case a SQL query. Systems for object-relational mapping and persistence also frequently use metaprogramming to generate or modify classes that load and store persistent data. Runtime metaprogramming has also been applied to generating code for generalized joins [14].

The current prototype of safe query objects uses the OpenJava [23] macro system to translate from the safe query class to a database access class. The translation from safe query class to database access class is done using compile-time meta-programming.

In OpenJava a compile-time transformation is specified by a *metaclass.* OpenJava extends the Java language so that a class definition can specify a metaclass using the `instantiates` keyword. The class `A` in Figure 2.2 instantiates the metaclass `M`, which is shown in Figure 2(b). The metaclass `M` adds a public field *x*`_str` of type `java.lang.String` for every field *x* in class `A`. The generated java class is shown in Figure 2(c). The method `translateDefinition` in metaclass `M` contains the code to translate from the source to the destination. In this example, the method calls `getField()` and `fields[i].getName()` are used for reflection and the method call `addField` is used to add new fields in the destination class.

Macros in OpenJava are compile-time transformations on class definitions and expressions that reference them. Like

```
interface javax.jdo.PersistenceManager {
    Object getObjectById(Object id);
    javax.jdo.Query newQuery(Class class);
    // methods for transactions not listed
}

interface javax.jdo.Query {
    void setFilter(String filter);
    void setOrdering(String ordering);
    void declareImports(String imports);
    void declareParameters(String params);
    void declareVariables(String vars);
    Object execute();
    Object executeWithMap(Map map);
    // bookkeeping methods not listed
}
```

Figure 3: Selections from the JDO API

Lisp macros, OpenJava macros operate on the abstract syntax tree of class definitions and expressions. Unlike Lisp, they cannot specify arbitrary input languages embedded in a generic syntax. OpenJava macros run after initial type analysis, but before complete type-checking has been performed.

2.3 Object-Relational Mapping and Persistence

Object-relational mapping [13, 6, 21] and persistent object systems [2, 21, 5] allow programmers to manipulate objects that are bound to persistent storage. This approach provides a uniform semantic view of all data as objects; persistent objects are loaded by *data shipping* as needed. These systems solve almost all of the problems of call level interfaces, but they have not displaced CLI because they do *not* support the one essential benefit of a call level interface: remote execution of high-level operations. Some systems [6, 5, 21] do support remote execution, but do so via a form of CLI.

Java Data Objects (JDO) is a new standard for interfacing Java with persistent data in relational and non-relational data stores [21]. JDO is a hybrid that supports both object-relational mapping and a call level interface. JDO uses a bytecode enhancer as a post-compilation step to transform ordinary classes into persistence-capable classes. The enhancer also adds code to track navigation and modifications to instances so that they can be loaded from and written to the persistent store as needed.

JDO provides access to persistent objects through an instance of the `PersistenceManager` interface shown in Figure 3. Individual objects can be loaded with `getObjectById`, while `newQuery` creates a JDO `Query` object, which is a call level interface. In JDO the call level interface has been broken into methods for filtering, ordering and declarations of query parameters, imports and variables. Filter conditions are written in the JDOQL language. The grammar of JDOQL is a subset of the Java grammar for expressions. However, the semantics of JDOQL are different from Java: JDOQL allows more automatic conversions between types, overloads `<` and `>` for boxed values including dates

```
class Employee {
  String     name;
  float      salary;
  Department department;
  Employee   manager;
}

class Department {
  String               name;
  Collection<Employee> employees;
}
```

Figure 4: Example classes

and strings, and interprets the `contains` method in a non-standard way to support existential quantification. The specific usage of each method is discussed as needed in the body of the paper.

3. FROM FILTER STRINGS TO SAFE QUERY OBJECTS

This section introduces the concept of safe query objects through a discussion of filter queries. The solution is motivated by an example filter query in JDO. A safe query object is then defined and the notions of local execution and remote database execution introduced.

Consider a simple database of employees and departments, whose schema is represented by the Java classes in Figure 4. A JDO query to select all employees whose salary is greater than their manager's salary is defined in Figure 5(a). This program is syntactically correct Java code, but there is a problem in the embedded code: `manager` is misspelled as `maneger`. The problem will not be discovered until run-time [9].

Every query has a *candidate type* that defines the class of objects returned by the query. For JDO, the candidate type is specified in the call to `newQuery`. The filter specification is evaluated for each instance of the candidate type: references to `salary` and `manager` access the members of the instance. JDO uses joins to implement navigation through object-valued fields, as in `manager.salary`. JDO ignores the access control restrictions on class members; in this paper all members are assumed to be public. The net effect of the query is to return the subset of all candidate instances in the database for which the filter is true.

JDO handles the translation of database rows into Java objects. Although the static return type of the `execute` method is `Object`, at runtime the return value is a read-only `Collection` containing instances of the candidate type. Although the current JDO specification does not make use of generic types in Java 1.5, the backward compatibility of generics allows JDO to work with generic classes. Generic types are used throughout this paper to increase precision of static typing for queries.

3.1 Safe Query Objects

This section introduces *safe query objects*, our solution to the impedance mismatch problem. Safe query objects support

```
Collection executePayCheck(
        javax.jdo.PersistenceManager pm)
{
  javax.jdo.Query payCheck =
      pm.newQuery(Employee.class);
  payCheck.setFilter("salary > maneger.salary");
  Object result = payCheck.execute();
  return (Collection) result;
}
```

(a) *Filtering in JDO (with runtime error)*

```
class PayCheckQuery extends SafeQuery<Employee> {
  boolean filter(Employee emp) {
    return emp.salary > emp.manager.salary;
  }
}
```

(b) *Safe query object*

Figure 5: Filtering example

static typing and remote execution of database queries. The approach uses local execution as a *reference semantics* to define the behavior of queries operating entirely within the Java language model.

In its simplest form, a safe query object is just an object containing a boolean method that can be used to filter a collection of candidate objects. A safe query implementing the `PayCheck` query is given in Figure 5(b). Because this filter is normal Java code, syntax and types are checked at compile time: if `manager` is misspelled, a compile time error is produced. The behavior of a query object is characterized by the `SafeQuery` base class:

```
class SafeQuery<T> {
  boolean filter(T item) { return true; }
}
```

The generic type parameter `T` is the candidate type of the query. The default implementation of `filter` returns true to include all candidate objects in the result set. Safe query objects are instances of *safe query classes*, which must extend `SafeQuery<T>`. In addition safe query classes must satisfy several behavior restrictions. For example, the filter method must be free of *side-effects*: they must not modify any state or call any methods that modify state. The full list of restrictions on safe query classes is given in Section 5.

3.2 Local Execution

Since queries are Java classes, they can be executed locally to filter any collection of objects, including objects shipped from a database. Local execution of a filter query is provided by the `execute` method in the class `Local` defined in Figure 6(a). This class assumes that all objects are local Java instances, and that the members of the candidate type are stored in a `Collection`.

```
class Local {
  <T> Collection<T> execute(SafeQuery<T> query,
                            Collection<T> candidates)
  {
    Collection<T> result = new ArrayList<T>();
    for (T item : candidates)
      if ( query.filter(item) )
        result.add(item);
    return result;
  }
}
```

(a) *Class implementing local execution*

```
Local local = ...; // load local database
Collection<Employee> emps = local.getEmployees();
PayCheckQuery query = new PayCheckQuery();
Collection<Employee> r = local.execute(query, emps);
```

(b) *Executing a query locally*

Figure 6: Local execution

The local execution of a query object is illustrated in Figure 6(b). The method for loading the set of employees is not defined. They could be stored in memory or loaded by shipping data from a database. Given that set, it executes the query to create a filtered list of employees.

Defining a query as a filter method is natural: it is a simple and effective Java programming pattern. There is no magic or special syntax involved. The pattern may be simple, but it modularizes query behavior into methods that have exactly the right form to be translated for remote execution in a relational database.

3.3 Remote Execution

The key to safe query objects is the mechanism for translating them into relational queries. To do so, the behavior specified in the filter method must be translated into an equivalent relational query.

For remote execution, compile-time metaprogramming is used to generate additional methods and attributes that enable the query to be shipped to a database for execution. The behavior for compile-time metaprogramming is encapsulated in the metaclass `RemoteQueryJDO`. This metaclass is applied to the safe query using the `instantiates` keyword of OpenJava as shown in Figure 7(a). OpenJava runs the metaclass at compile time, supplying the definition of the `PayCheck` class as an input. The metaclass can examine the partially-compiled definition of a class and modify or extend the class.

When applied to `PayCheck`, the `RemoteQueryJDO` metaclass generates the `execute` shown in Figure 7(b). This method implements a remote version of the `PayCheck` filter method by passing appropriate strings to the JDO interface. The generated method is the same as the original unsafe code in Figure 5(a). However, because the code is generated automatically from a type-checked Java method, the safe query version is type safe. The details of how OpenJava is used to perform the translation are given in Section 5

```
class PayCheck instantiates RemoteQueryJDO
               extends SafeQuery<Employee>
{
  boolean filter( Employee emp ) {
    return emp.salary > emp.manager.salary;
  }
}
```

(a) *Safe query class that invokes remote metaclass*

```
Collection<Employee> execute(
        javax.jdo.PersistenceManager pm )
{
  javax.jdo.Query q = pm.newQuery(Employee.class);
  q.setFilter( "salary > manager.salary" );
  return (Collection<Employee>) q.execute();
}
```

(b) *Automatically generated method for PayCheck*

```
javax.jdo.PersistenceManager pm;
PayCheck query = new PayCheck();
Collection<Employee> r = query.execute(pm);
```

(c) *Using a query for remote execution*

Figure 7: Remote execution

Execution of a remote query is shown in Figure 7(c). The safe query provides a statically-typed interface to the JDO implementation.

The relationship between local execution and remote execution is central to the definition of safe query objects. Ideally the semantics of these two execution strategies should be identical; however, this is not absolutely essential – if at least the semantics of remote execution is precisely defined. The key requirement is that the queries be type-checked statically while still supporting remote execution in the database engine. This is a particular issue with Java and SQL, because they have very different interpretations of NULL values. This issue is discussed in more detail in Section 6.

4. ADDITIONAL FEATURES OF SAFE QUERY OBJECTS

The previous section described the simplest kind of queries that have a boolean method to filter objects from the extent of a class. Real-world applications require more features for the creation and specification of queries. The following sections discuss sorting, parameterization, dynamic queries, and existential quantification.

4.1 Sorting Results

Queries often specify a *sort order* for the results that match a filter. Relational query languages define sort order by deriving a list of sortable values from each element of the candidate type. The list of values is annotated to indicate whether the sort should be in ascending or descending order. For example, in SQL the list of values is given in the `ORDER BY` clause. A similar approach is used in JDO, as shown in Figure 8(a). The first value in the list defines the primary order; only when the first values are equal are subsequent values considered.

Note that this approach to sorting is different from the one used in most data structure libraries, including the Java Collection classes. Data structure libraries usually define sort order by a comparison function. One advantage of a comparison function is that avoids generating a list of sortable values, which immediately become garbage after being used. One disadvantage is that a compiler cannot ensure that a general-purpose boolean function is a comparison function that defines a partial order; so comparison functions may be unsafe. The issue of extra garbage is only significant for query objects when executed locally.

To specify sorting, a safe query must associate a list of sortable values with each object in the result set. This is done by adding a `order` method to the query that takes a candidate element and returns a list of sortable values. The list of sortable values is represented as a linked list of `Sort` objects, each of which contains a value, a flag indicating ascending or descending order, and an optional secondary `Sort` value. An example safe query that specifies a sort order is given in Figure 8(b). The methods in the `Sort` class are:

```
class Sort implements Comparable<Sort> {
    Comparable v;    // sortable value
    Direction dir;   // direction
    Sort next;       // optional secondary sort
    enum Direction {ASCENDING, DESCENDING};
    Sort(Comparable v, Direction dir)...
    Sort(Comparable v, Direction dir, Sort next)...
    int compareTo(Sort other)...
}
```

It is possible to write an `order` method that is not well-behaved. For example, it might return lists with different lengths or containing different types of Comparable values. There does not appear to be any simple typing for this example in Java that prevents the possibility of runtime errors. The OpenJava metaclass can use conservative static analysis to reject such methods and signal a compile-time error.

Local execution is performed by using a `SortedSet` for the result collection and an appropriate `SortComparator` object to compute the lexicographic sorting of two objects' sort values. To support sorting, the execute method in Figure 6(a) is modified to use a sorted set for the result collection by replacing `ArrayList<T>()` with `TreeSet<T>(SortComparator)`. The sort comparator implements `compare(a, b)` by calling `order` on `a` and `b` and then comparing the resulting `Sort` objects.

Remote execution with sorting is similar to remote execution with filtering. Compile-time metaprogramming is used to

```
javax.jdo.Query q = pm.newQuery(Employee.class);
q.setOrdering("department.name ascending,"
          + "salary descending");
Collection r = (Collection) q.execute();
```

(a) *Hand-coded sorting with JDO*

```
class SortQuery instantiates RemoteQueryJDO
                extends SafeQuery<Employee>
{
  Sort order(Employee emp) {
    return new Sort(emp.department.name,
                    Sort.Direction.ASCENDING,
           new Sort(emp.salary,
                    Sort.Direction.DESCENDING));
  }
}
```

(b) *Safe query specifying sort order*

```
Collection<Department> execute(
        javax.jdo.PersistenceManager pm)
{
  javax.jdo.Query q = pm.newQuery(Employee.class);
  q.setOrdering("department.name ascending, "
            + "salary descending");
  return (Collection<Employee>) q.execute();
}
```

(c) *Automatically generated remote execution with sort order*

Figure 8: Sorting results

translate the sorting specification in the `order` method into the syntax accepted by JDO, as shown in Figure 8(c). The resulting code is nearly identical to the hand-written code, but because it is generated automatically from the statically-checked order method, it is type-safe. Of course, sorting and filtering can be used together in a query.

4.2 Parameterized Queries

Parameterized queries are needed when a query's behavior depends upon one or more input values. Executing parameterized queries with a call level interface is awkward because the essential semantic connections between the declaration, use, and binding of a parameter are broken up by the disjoint API calls in a call level interface. Figure 9(a) illustrates this situation in a JDO query to find employees whose salary is above a given limit. The parameter is used in the `setFilter` call, declared in the `declareParameters` call, and bound in `execute`. These calls can be made in any order, and the complex semantic constraints between them are not checked at compile time. There are many ways that this code can go wrong at runtime. For example, the call to `declareParameters` may be omitted, or the wrong type of value can be passed to `execute`. Again, these problems will only be discovered at run-time.

Safe queries could use a variety of Java features to implement query parameters. Query parameters are values that can affect the filtering and ordering of query results. One way to make them accessible from the `filter` and `order` methods is to include them as instance variables in the query object. Parameters on the class constructor are used to initialize the variables. Figure 9(b) defines a parameterized safe query class to find employees with salary greater than a limit. Because parameters are normal Java variables, their declaration, use, and binding are all checked for consistency at compile time.

The `RemoteQueryJDO` metaclass translates the query parameters into appropriate calls to JDO. The result, shown in Figure 9(c), is very similar to the hand-coded version discussed above. The way in which parameters are specified varies somewhat between different call level interfaces, but the overall pattern is similar. The dynamically generated version uses the JDO `executeMap` method to pass a dictionary containing all variable bindings. Because the translation is automated, there are guaranteed to be no syntax or type errors in the strings passed to CLI.

Both local and remote execution of a parameterized query are similar to execution of a simple query, except that parameters are supplied when the query is constructed. An example of remote execution is given below:

```
SafeQuery<Employee> q = new SalaryLimit(50000);
Collection<Employee> result = q.execute(pm);
```

4.3 Dynamic Queries

Dynamic queries involve filters, parameters, or sort orders that are constructed at runtime. They are used when different filter criteria must be combined to form a complete filter. For example, if a user interface allows a set of optional search criteria to be specified, the filters that result from different

```
javax.jdo.Query q = pm.newQuery(Employee.class);
q.setFilter("salary > limit");
q.declareParameters("Double limit");
Collection r = (Collection) q.execute(50000);
```

(a) *Hand-coded parameterized query (with errors)*

```
class SalaryLimit instantiates RemoteQueryJDO
                  extends SafeQuery<Employee>
{
  double limit;  /* parameter */
  SalaryLimit(double limit) {
    this.limit = limit;
  }

  boolean filter(Employee employee) {
    return employee.salary > limit;
  }
}
```

(b) *Safe query object with parameterization*

```
Collection<Employee> execute(
        javax.jdo.PersistenceManager pm)
{
  javax.jdo.Query q = pm.newQuery(Employee.class);
  q.setFilter("salary > limit" );
  q.declareParameters("double limit");
  Map paramMap = new HashMap();
  paramMap.put("limit", limit); // boxed
  Object result = q.executeWithMap(paramMap);
  return (Collection<Employee>) result;
}
```

(c) *Automatically generated code*

Figure 9: Parameterized queries

combinations of criteria will be different. Since the difference between the filters is structural, parameterized queries alone are not sufficient.

Dynamic filters are commonly created by concatenating portions of a filter string together to create the complete filter. If the different filter components have parameters, then the set of parameters to the combined query is also dynamic. Figure 10(a) illustrates creation of dynamic filters and parameters using JDO. In this example, a user can search for employees by name, salary range, or both. Different filters are constructed depending on which criteria the user specifies.

Safe query objects implement dynamic filters as a special case of normal filters. A dynamic filter can be expressed as a filter method in Java through the use of conditionals or short-circuit evaluation: parts of the overall filter are only evaluated if certain conditions are met. Figure 10(b) illustrates this technique for the dynamic query given in Figure 10(a). Short-circuit evaluation of || is required to avoid null-pointer exceptions. Local execution of a dynamic query is the same as a normal parameterized query. This query can be used to find employees whose name begins with “F”, but with no limit on salary:

```
GenericQuery q = new GenericQuery("F", null)
Collection<Employee> = q.execute(pm);
```

Dynamic filters are sometimes used to replace explicit parameters. The parameter values are embedded into the query expression using string concatenation. This approach was illustrated in Figure 1 for a double value, but can also be used for string parameters, although proper quoting becomes a serious issue. Creating a filter by concatenation, as in `"name=\"" + testName + "\""` is notoriously unsafe and fragile. If the variable `testName` contains a quote, the query will fail at runtime, or open the possibility for executing arbitrary code on the database server. Using dynamic filters can lower performance because the database query optimizer can reuse a query plan if the submitted query is identical to a previous query.

Although Figure 10(b) looks like an ordinary parameterized query class, it cannot be translated to a single JDO filter for remote execution using the techniques given above. One reason for this is that call level interfaces and databases do not typically support *null* parameters; for example, JDBC does not. In addition, the SQL standard does not require short-circuit evaluation of boolean connectives, so the query may generate incorrect results.

The key issue is that parts of the filter condition depend only on local data, and so can be evaluated before being sent to the database. For example, the expression `namePrefix == null` depends only upon a parameter value, while `emp.salary >= minSalary` depends upon remote data. Abstract partial evaluation is used to find the cases where a null argument triggers short-circuit evaluation. These cases are then generated separately. Ternary if expressions `c ? t : f` are also handled, although due to limitations in JDO the conditional expression must only refer to local data (parameters).

```
Collection search(String namePrefix,
                  Double minSalary)
{
  String filter = null;
  String paramDecl = "";
  HashMap paramMap = new HashMap();

  if (namePrefix != null) {
    q.declareParameters("String namePrefix");
    paramMap.put("namePrefix", namePrefix);
    filter = and(filter,
                 "(name.startsWith(namePrefix))");
  }

  if (minSalary != null) {
    q.declareParameters("double minSalary");
    paramMap.put("minSalary", minSalary);
    filter = and(filter,
                 "(salary >= minSalary)");
  }

  javax.jdo.Query q = makeQuery(Employee.class);
  q.setFilter(filter);

  return q.executeWithMap(paramMap);
}

String and(String a, String b) {
  return (a == null) ? b : (a + " && " + b);
}
```

(a) *Construction of a dynamic filter in JDO*

```
class DynQuery instantiates RemoteQueryJDO
               extends SafeQuery<Employee>
{
  private String namePrefix; // may be null
  private Double minSalary;  // may be null

  DynQuery(String namePrefix, Double minSalary) {
    this.namePrefix = namePrefix;
    this.minSalary = minSalary;
  }

  boolean filter(Employee item) {
    return (namePrefix == null
             || item.name.startsWith(namePrefix))
        && (minSalary == null
             || item.salary >= minSalary);
  }
}
```

(b) *Safe query using dynamic filter*

Figure 10: Dynamic filters

Currently the analysis only supports dynamic queries based on tests for null. The resulting code is very similar to the hand-written program given in Figure 10(a). Details of the implementation are given in Section 5.

4.4 Existential Quantification

The SQL query language supports existential quantification to test if any member of a set meets a condition. For example, a query may find all departments that have an employee who meets some criteria. This requires the employee criteria to be added to an existential clause in the department filter. It is desirable to reuse the code that creates filter criteria. However, using JDO this could only be done by string concatenation.

Most object-oriented programming languages do not have standard syntax for existential quantification, but some collection libraries include methods to test if any member of a collection satisfies a condition. In Smalltalk the method is called `detect:ifNone:`. This behavior is added to the `SafeQuery` class as an `exists` method, as defined in Figure 11(b).

Note that this definition of exists is based on *reuse* of other query objects as subqueries. A filter using existential quantification is given in Figure 11(a). It finds departments whose name begins with a given prefix and have an employee whose salary is above a given minimum. The query reuses the `SalaryLimit` query defined in Figure 9(b).

A translation of this query into JDO is given in Figure 11(c). This translation is a simplified version of the full translation presented in the next section. JDO expresses existential quantification by interpreting the `contains` method as a binding operator for free variables, which must be declared in a call to `declareVariables`. The filter is true if there is a substitution of the free variables for objects that satisfies the filter expression. The query in Figure 11(c) is a reasonable illustration of the kind of complexity that programmers face in writing even fairly simple queries. The two levels of interpretation are clearly visible: Java code that manipulates JDO expressions as strings.

5. IMPLEMENTATION DETAILS

In the prototype implementation of safe query objects, OpenJava is used as a framework for compile-time metaprogramming and method generation. *Reflection* in java is the ability for a program to gain access to its own code. The system is implemented as a *metaclass*, which uses reflection to examine the query class and generate methods for remote execution. Figures 12(a) and 12(b) show a high-level view of the derivation of the `execute` method from a query class. Figure 12(a) specifies a pattern that identifies the components of the query class used in the derivation. Repeated elements are identified by ⟨ ... ⟩ together with an index. Figure 12(b) defines the template for generating the remote execute method, based on the bindings of variables in the pattern.

Most of the template is a direct translation of the Java code into corresponding embedded code. However, the treatment of subqueries is more complex. When compiling a complete program, it would be possible to build a static string repre-

```
class DeptQuery instantiates RemoteQueryJDO
                extends SafeQuery<Department>
{
  double min;
  String deptPrefix;

  DeptQuery(double min, String deptPrefix) {
    this.min = min;
    this.deptPrefix = deptPrefix;
  }

  boolean filter(Department dept)
  {
    SafeQuery<Employee> sub = new SalaryLimit(min);
    return dept.name.startsWith(deptPrefix)
           && exists(dept.employees, sub);
  }
}
```

(a) *Existential quantification in a safe query class*

```
class SafeQuery<T> {
  boolean filter(T item) { return true; }
  Sort    order(T item)  { return null; }
  boolean exists(Collection<T> candidates,
                 SafeQuery<T> query) {
    for (T item : candidates) {
      if ( query.filter(item) )
        return true;
    }
    return false;
  }
```

(b) *Complete definition of SafeQuery with exists*

```
Collection<Employee> execute(
        javax.jdo.PersistenceManager pm )
{
  javax.jdo.Query q = pm.newQuery(Employee.class);

  HashMap paramMap = new HashMap();
  paramMap.put("min", min);
  paramMap.put("namePrefix", namePrefix);
  q.declareParameters("double min");
  q.declareParameters("String namePrefix");

  q.declareVariables("Employee e");
  String filter = "name.startsWith(deptPrefix)"
              + "  && (employees.contains(e)"
              + "       && e.salary > min)");
  q.setFilter(filter);

  Object r = q.executeWithMap(paramMap);
  return (Collection<Employee>) r;
}
```

(c) *JDO version of existential quantification and parameters*

Figure 11: Existential quantification

```
$\langle$ import $import_m$; $\rangle_m$
class $QueryName$ instantiates RemoteQueryJDO
                 extends SafeQuery<$ResultType$>
{
  // all members are parameters:
$\langle$ $ParamType_i$ $param_i$; $\rangle_i$

  boolean filter($ResultType$ $elem$) {
    return $filter$;
  }
  Sort order($ResultType$ $orderVar$) {
    return new Sort($order_j$,
                Sort.Direction.$dir_j$,
                $sort_{j+1}$ );
  }
}
```

(a) *Pattern matched against class definition*

```
// constructor for $QueryName$ class
$QueryName$($\langle$ $ParamType_i$ $param_i$; $\rangle_i$) {
$\langle$ this.$param_i$ = $param_i$; $\rangle_i$
}

// remote execution method
Collection<$ResultType$> execute(
javax.jdo.PersistenceManager pm)
{
  javax.jdo.Query q =
        pm.newQuery($ResultType$.class);
  Map paramMap = new HashMap();
$\langle$ paramMap.put("$param_i$", $param_i$ ); $\rangle_i$
$\langle$ q.declareParameters("$ParamType_i$ $param_i$"); $\rangle_i$

  q.setFilter( $\Phi(elem,\ filter)$ );
  q.setOrdering($\langle\Phi(elem,\ order_j)$ + " $dir_j$,"$\rangle_j$);

  Object result = q.executeWithMap(paramMap);
  return (Collection<$ResultType$>) result;
}

// subquery with variable substitutions
static String subquery(String $elem$,
                    $\langle$ String $param_i$, $\rangle_i$) {
  return $\Theta$( { $elem$, $\langle$ $param_i$ $\rangle_i$ }, $filter$ );
}

// exists query and JDO declarations
static String makeExists(javax.jdo.Query q,
                    $\langle$ String $param_i$, $\rangle_i$) {
  String $elem$ = CreateUniqueName();
  String decl = "$ResultType$ " + $elem$;
  q.declareVariables(decl);
  q.declareImports("$\langle$ import $import_m$; $\rangle_m$");
  return "contains(" + $elem$ + ") && "
         + subquery($elem$, $\langle$ $param_i$ $\rangle_i$) );
}
```

(b) *Template for generated methods*

Figure 12: Outline of translation

senting the elaboration of all subquery behavior. However, this would preclude the possibility of separate compilation and reuse of query libraries. As a result, the translation given here requires dynamic construction of JDO expressions that involve subqueries. If no subqueries are used, then the filter strings are all static.

5.1 Translating Expressions

The function Φ converts a Java expression e into a Java expression that creates a string representation of e in JDOQL. The reason that Φ must return a Java expression instead of a string is any subqueries require calls to appropriate methods of the subquery object to assemble a full expression. The first argument of Φ is the name of the element variable (the formal parameter of the filter method). In JDOQL the current element being filtered is implicit, so Φ must strip off uses of this variable from the Java code. The second argument is the Java expression syntax being translated. The function Φ is defined as follows.

$$\Phi : Variable \times JavaExpr \rightarrow JavaExpr$$
$$\Phi(v,\ v) \rightarrow \texttt{""}$$
$$\Phi(v,\ x) \rightarrow \texttt{"}x\texttt{"}$$
$$\Phi(v,\ e.f) \rightarrow \Phi(e,\ e)\ \texttt{+ ".}f\texttt{"}$$
$$\Phi(v,\ \texttt{exists(e, new } Q(\langle\ \texttt{a}_i,\ \rangle_i))) \rightarrow \Phi(v,\ \texttt{e})\ \texttt{+ "." + } Q\texttt{.makeExists(q, } \langle\Phi(v,\ \texttt{a}_i)\rangle_i)$$

For the most part, the syntax of JDOQL resembles Java syntax, so most expressions are unchanged. JDO uses `<` and `>` for date comparison instead of `before` and `after` methods. Existentials are converted to JDO syntax.

The variable *elem* can only appear in a field access expression. Φ returns a string constant except when the expression contains a call to `exists`. In that case Φ generates a call to the static `makeExists` method of the subquery Q, passing the local JDO query `q` and translation of the expressions $\texttt{a}_i$ used to initialize the subquery.

5.2 Translating Existentials

The static `makeExists` method is defined in the template in Figure 12(b). It creates a new variable name to represent the quantified variable (similar to a skolem constant). It then calls the query `q` to declare the new variable and any query imports. Finally it returns a new `contains` expression and generates the query text by calling the `subquery` method.

The static `subquery` method creates a version of the filter expression with parameters substituted by the actual parameter expressions from the main query. The transformation Θ, which performs this renaming, is a variation on Φ:

$$\Theta : Set(Variable) \times JavaExpr \rightarrow JavaExpr$$
$$\Theta(s,\ x) \rightarrow if\ x\ \in\ s\ then\ x\ else\ \texttt{"}x\texttt{"}$$
$$\Theta(s,\ e.f) \rightarrow \Theta(s,\ e)\ \texttt{+ ".}f\texttt{"}$$
$$\Theta(s,\ \texttt{exists(e, new } Q(\langle\ \texttt{a}_i,\ \rangle_i))) \rightarrow \Theta(s,\ \texttt{e})\ \texttt{+ "." + } Q\texttt{.makeExists(q, } \langle\Theta(s,\ \texttt{a}_i)\rangle_i)$$

Only the first rule of Θ differs significantly from Φ. It specifies that all query variables x contained in the set s are replaced by the dynamic contents of the Java variable x, while all other variables are replaced by a string constant containing the variable name. The set of variables passed to Θ includes the element variable and all parameters of the query. In the subquery and makeExists methods, these variables are redefined as String variables whose contents are concatenated to create the subquery. As an example, here is the translation of the SalaryLimit class:

```
static String subquery(String emp, String limit)
{
  return emp + ".salary > " + limit;
}

static String makeExists(javax.jdo.query q,
                          String limit)
{
  String emp = CreateUniqueName();
  String decl = "Employee " + emp;
  q.declareVariables(decl);
  return "contains(" + emp + ") && "
           + subquery(emp, limit);
}
```

Figure 11(a) used SalaryLimit as a subquery in the form exists(d.employees, new SalaryLimit(min)). An approximate translation was given in Figure 11(c), but a correct modular translation can now be given:

```
"(employees."+ SalaryLimit.makeExists("min") +")"
```

At runtime this call will generate a new variable name v and return this expression:

```
(employees.contains(v) && v.salary > min)"
```

The current transformation does not support remote execution of queries whose parameters are objects, although such queries can be used in existential quantification. Future work will extend the prototype to support object parameters by substituting each expression containing an object parameter with a synthetic parameter representing the value of the expression.

5.3 Translating Dynamic Queries

Dynamic queries discussed in Section 4.3 require an additional step during translation. The problem is that the query condition is not a strict boolean expression, instead it uses conditional logic expressed by short-circuit evaluation of ||. The conditional logic tests whether parameters to the query are null. We use the following approach for interpreting null values – if the expression being evaluated can be simplified given the subset of the parameters that are null, then the original expression is replaced by its simplified form. For example, in java the expression (d == null) || (d.salary > 50) simplifies to true when d is null and d.salary > 50 when d is not null. Similarly the return expression in example of Figure 10(b) can be simplified as follows.

```
namePrefix == null, minSalary == null
   → true
namePrefix != null, minSalary == null
   → item.name.startsWith(namePrefix)
namePrefix == null, minSalary != null
   → item.salary >= minSalary
namePrefix != null, minSalary != null
   → item.name.startsWith(namePrefix) &&
      item.salary >= minSalary
```

Simplification of queries is done using *abstract partial evaluation* [19], an evaluation strategy which can partially evaluate an expression given some abstract property of the variables used in the expression. The abstract property we are interested in is whether a variable is null. Given this abstract property, boolean expressions such as x == y, x != y, x && y and x || y can be simplified. Since the return value of a filter is a boolean expression this level of abstraction is sufficient.

The query that is shipped to the database is parameterized by the subset of parameters in the safe query class that are not null. The set of actual parameters to the query which are specified in the call q.declareParameters() is the subset of non-null values in the parameter set of the safe query object.

We use abstract partial evaluation to simplify the expression being used in the filter method. The expression Φ(*elem*, *filter*) is simplified for each subset of the parameters that can be null. We use a switch statement at run-time, switching on the subset of parameters that are null, to choose the appropriate expression. For ease of exposition the code templates shown in this section do not contain details about abstract partial evaluation.

6. EVALUATION AND FUTURE WORK

The current design of safe query objects is based on a set of principles

- Query objects must be standard Java objects. It must be possible to compile and execute them locally without any support by compile-time metaprogramming.
- To the degree possible, a statically-typed query object should execute locally and remotely without runtime errors.
- If necessary, compile-time metaprogramming can place restrictions on the range of code allowed in a query object. For example, metaclasses can prevent the use of assignment statements, but should not alter their meaning.
- The behavior of local and remote execution of queries should be identical. The current design does not achieve this goal because it is very difficult to modify the way that SQL handles NULL, and this behavior is quite reasonable for use in filters. In the current implementation the behavior of local and remote execution are not identical; the best solution is to use compile-time metaprogramming to modify the local filter method so that its behavior matches SQL semantics.

The transformation does not guarantee identical semantics for local and remote execution. In SQL, comparison with null values is defined to always be false: null is not less than, greater than, equal to, or not equal to any value. In addition, JDO specifies that navigation through a null-valued field, which would throw `NullPointerException`, is treated as if the filter expression returned `false`. It should be possible to modify the local filter method to conform to the behavior of remote execution by wrapping each Boolean-valued term in the filter expression with a try-catch block that translates `NullPointerException` to `false`.

The query definitions currently require all class members used in filters to be public. This is reasonable for classes associated with database tables, based on the value object pattern [12]. Accessor methods could be supported by looking up the associated field name with compile-time reflection.

JDO is useful because it automates mapping of relational tables to classes and generates the joins necessary to access related objects in filters. However JDO has a number of significant limitations. Although joins can be used for filters, the result of a JDO query is always a set of objects of a single type – no related objects can be returned at the same time. Several aspects of SQL are not supported, including general aggregation and `case` expressions. Future work on safe query objects will remove the dependence on JDO.

The current version of OpenJava does not support generics, so the generic type erasing has been done manually in the prototype implementation.

7. RELATED WORK

HaskellDB [15] is a SQL library for Haskell. It defines an abstract data type for database queries that includes operations for selections, joins and filtering. Queries are written as monadic list comprehensions – however, rather than performing the operations on local Haskell data, the monad instead builds an internal representation of a query. The resulting abstract queries are converted into SQL for remote execution on a database. To use HaskellDB, the programmer must define the structure of the database as a set of Haskell types. Given these, the HaskellDB queries are all statically typed. HaskellDB works without any special reflective support; instead the query sublanguage is embedded within Haskell yet is indistinguishable from standard Haskell operations.

The current version of HaskellDB does not support existential quantification or sorting. Parameterized queries are easily defined because HaskellDB is embedded cleanly within the general-purpose Haskell language. However, the queries are generated with parameters embedded in the query text rather than passed via the CLI parameter interfaces. As mentioned in Section 4.3, this can reduce performance because dynamic queries must all be processed separately, while parameterized queries can use a cached query plan. The query strings are also generated at runtime rather than compile time, although this is likely to have only a small impact on performance.

SchemeQL [25] is a SQL library for Scheme. It uses macros to generate data structures representing SQL queries, which can be nested and composed easily. The macros define new syntactic forms that resemble SQL queries, rather than reusing existing language syntax and semantics as in query classes. SchemeQL does not support existential quantification. It does not support parameterized queries, although they can be created by defining Scheme functions that return queries; as in HaskellDB, the parameters are embedded in the resulting SQL code, rather than being explicit query parameters. As mentioned above, this affects database performance; because every query string is different, the database query optimizer cannot reuse plans created for previous queries. The design of SchemeQL has a few fairly serious issues. For example, the macros cannot distinguish symbols representing strings and constants from variables, so the user of the library must quote string literals properly using the quoting conventions of the underlying database, not of Scheme.

Brant and Yoder [4] present patterns for creating database-reporting applications, which often require complex and dynamic queries. Their *Composable Query Object* pattern specifies a dynamic object representation of queries. Subclasses of *QueryObject* include *TableQuery* for accessing tables, *JoinQuery* for merging tables, *SelectionQuery* for filtering results, and *ProjectQuery* for specifying query outputs. Other subobjects are used to represent the details of a query. For example, a *SelectionQuery* contains an object representation of the selection condition. *Query Objects* are an object-oriented representation of the relational calculus with appropriate operations for creating, composing, and executing queries. The result is a sublanguage embedded within the target language, Smalltalk, although the sublanguage has different scope and execution semantics from Smalltalk. *Query Objects* are very similar to SchemeQL, although more work is needed to determine their exact relationship.

Gould, Su and Devanbu [8] take a different approach to type-checking programs with embedded SQL. They perform static analysis to identify strings that contain embedded SQL code. The construction of these strings is tracked so that they can be checked against the SQL grammar and typing rules. Their analysis does not currently cover query parameters or result types. In addition, it generates many more false positives when separate compilation is used. Type-checking embedded SQL is a pragmatic solution, given that many existing program can benefit from more static analysis. However, the programming model of using call level interfaces is still complex, and programmers must work with two different languages at the same time. Safe query objects provide an alternative that simplifies access to relational development, provides a single uniform programming model, and scales well to separate compilation.

Techniques have also been introduced for optimizing queries by shipping data or operations to the location where they can be executed most efficiently [20]. In this case the operations are already expressed in a relational query language, while the problem addressed in this paper is separating out operations specified in object-oriented languages.

8. CONCLUSION

This paper introduces a new way to overcome the impedance mismatch between object-oriented programming languages and relational databases. The fundamental problem is how to provide a semantically integrated model of persistent objects and behavior, while still allowing large parts of the program to be "picked out" for efficient execution within a database engine. We introduce a natural programming pattern for expressing queries as objects; methods are used to express query selection criteria and the sort order of results. Class members/constructor arguments are query parameters. Queries can be reused directly or for existential quantification. We also provide a new analysis of conditional criteria, also known as dynamic queries, and a clean implementation using abstract partial evaluation.

Safe query objects support query shipping so that they can be executed efficiently within a database engine. Compile-time metaprogramming is used to augment safe query objects with methods that send corresponding queries to a relational database for execution. We have developed a prototype implementation using OpenJava and generate code for JDO, a persistence library for Java.

Acknowledgments

This work grew out of a course on Integrating Programming Languages and Databases at UT in the fall of 2003. We would like to thank the participants of that course, and especially Kevin Loo, who worked on an initial design and implementation of this approach. Thanks to Shriram Krishnamurthi, Dan Miranker, Mark Grechanik, Dave Thomas, Ralph Johnson and Joe Yoder for their comments on early drafts of this paper.

9. REFERENCES

[1] American National Standard for Information Technology. Database languages - SQLJ - Part 1: SQL routines using the Java programming language. Technical Report ANSI/INCITS 331.1-1999, InterNational Committee for Information Technology Standards (formerly NCITS), 1999.

[2] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[3] T. Bloom and S. B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 441–451. ACM Press, 1987.

[4] J. Brant and J. W. Yoder. Creating reports with query objects. In B. F. Neil Harrison and H. Rohnert, editors, *Pattern Languages of Programs Design 4*. Addison Wesley, 2000.

[5] D. Cengija. Hibernate your data. *onJava.com*, 2004.

[6] J.-A. Dub, R. Sapir, and P. Purich. Oracle Application Server TopLink application developers guide, 10g (9.0.4). Oracle Corporation, 2003.

[7] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 149–160. ACM Press, 1996.

[8] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings, 26th International Conference on Software Engineering (ICSE)*. IEEE Press, 2004.

[9] M. Grechanik, D. Perry, and D. Batory. An approach to evolving database dependent systems. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 113–116. ACM Press, 2002.

[10] G. Hamilton and R. Cattell. JDBC^{TM}: A Java SQL API. Sun Microsystems, 1997.

[11] W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *Proc. of the fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 239–248. ACM Press, 1993.

[12] Java Center. J2EE^{TM} patterns. Sun Microsystems, 2003.

[13] W. Keller. Mapping objects to tables - a pattern language. In *Proceedings of the 1997 European Pattern Languages of Programming Conference*, number 120/SW1/FB in Siemens Technical Report, Irrsee, Germany, X. EA Generali, Vienna, Austria, .

[14] G. Kirby, R. Morrison, and D. Stemple. Linguistic reflection in Java. *Software–Practice and Experience*, 28(10):1045–1077, 1998.

[15] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122. ACM Press, 1999.

[16] D. Maier. Representing database programs as objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1, September 1987, Roscoff, France*, pages 377–386. ACM Press / Addison-Wesley, 1987.

[17] V. Matena and M. Hapner. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.

[18] Microsoft Corporation. OLE DB/ADO: Making universal data access a reality, 1998.

[19] G. Puebla and M. Hermenegildo. Implementation of multiple specialization in logic programs. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 77–87. ACM Press, 1995.

[20] M. Rodriguez-Martinez and N. Roussopoulos. MOCHA: a self-extensible database middleware system for distributed data sources. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 213–224. ACM Press, 2000.

[21] C. Russell. Java Data Objects (JDO) Specification JSR-12. Sun Microsystems, 2003.

[22] T. Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer-Verlag, 2001.

[23] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for Java. In *OORaSE'99 Workshop on Object-Oriented Reflection and Software Engineering*, pages 117–133, Denver, Colorado, USA, Nov. 1999. ACM.

[24] M. Venkatrao and M. Pizzo. SQL/CLI - a new binding style for SQL. *SIGMOD Record*, 24(4):72–77, 1995.

[25] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Third Workshop on Scheme and Functional Programming*, 2002.