STRONGLY TYPED, COMPILE-TIME SAFE AND LOOSELY COUPLED DATA PERSISTENCE

by

CONGLUN YAO

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
July 2010

Abstract

A large number of approaches have been developed to simplify construction of, and to reduce errors in, data-driven applications. However, these approaches have not been particularly concerned with compile-time type safety. Type mismatch errors between program and the database schema occur quite often during program development, and the techniques used in these approaches often defer error checking on database operations until runtime.

In this thesis, we take a different approach from those previously proposed, based on strict type checking at compile time, type inference, higher-order functions, phantom types, object relational mapping, and loosely coupled database interaction. Instead of using external, literal XML file and string type SQL, we embed the mapping meta data and user defined queries directly in the program, the type safety of which is guaranteed by the program compiler. Such a result is achieved by introducing additional database schema information and using type *avatars*, a dummy structure used to extend the type checking to embedded queries, during compilation.

We show that this approach is practical and effective by implementing a compile-time type-safe object relational framework, called Qanat, in the OCaml programming language and using a loosely coupled SQL database. We further report experimental results obtained by running a number of benchmark tests, and compare the resulting Qanat applications with the equivalent, raw database driver based applications.

Acknowledgements

I would like to thank my supervisor, Alan P. Sexton, for his unstinting support throughout my PhD, and the many friends I have made during my time in Birmingham — you know who you are.

I owe my loving thanks to my wife, Duoduo, and our families, without the encouragement of whom I would have found it impossible to rise to this level.

Contents

List of Figures

Chapter 1

Introduction

Data-driven applications largely rely on external systems for the storage and retrieval of persistent data. Because they provide efficient and flexible querying and indexing, reliable transaction and concurrency control, and easy disaster recovery, relational databases have become the predominant approach for data persistence over the past few decades and are likely to remain so for at least the next few years. However, because of different programming paradigms between relational database management systems and object oriented programming languages, referred to as the object-relational impedance mismatch problem [CM84, Amb03b], it is surprisingly difficult to quickly develop reliable, well-engineered database applications in object oriented programming languages. To date, a number of approaches have been proposed to make such integration simpler and type-safe. In this thesis, we survey previous approaches and describe and discuss our new approach to this problem.

1.1 Motivation

Relational databases, as implemented in relational database management systems (RDBMS), store and manage a collection of data structured as tables, based on the relational model [Cod83], which was introduced by E.F.Codd in 1970 and developed on the basis of predicate logic and set theory. Based on these mathematical theories, the relational database has three well-defined components [RC93]: a logical data structure represented by the relational table for data storage, a set of integrity rules enforcing the data to be and remain consistent over time, and a set of operations for data querying and manipulation. In addition, it encompasses features such as indexing for efficient querying, a transaction mechanism for grouping operations for complete commit or rollback, concurrency control for data sharing, and disaster recovery for data reliability. The relational database is viewed as loosely coupled to the application program, but allowing the maintenance of persistent program data beyond the life-cycle of applications. Relational databases, based upon these features, are typically used for the storage of financial records, manufacturing and personnel information and much more.

The term "object-relational impedance mismatch" [CM84, Amb03b] denotes the problem that the different paradigms of object oriented programming languages and relational databases are sufficiently different to cause difficulties in bridging the gap between them. The programming language is based on proven software engineering principles, and supports the building of applications out of objects that encapsulate both data and behaviours, in the case of object-oriented programming languages. While the relational paradigm is based on proven mathematical principles, it persists data in tables and manipulates that data through a language based on relations, tuples and attributes [CB74].

Programs typically communicate with databases through the *call-level interface* (CLI) [VP95, ISO]; that is, the program constructs queries in the form of strings using string concatenation operation, then these unstructured query strings are shipped to the database server for execution. The database server parses, compiles and executes the query, then returns results as a string stream. If any errors happens during this phase, a runtime exception is thrown. The following code

demonstrates a typical call-level-interface query execution using the JDBC (Java Database Connection) [FEB03] library; basically, it involves query string concatenation, connection initialisation, query execution and result processing.

```
String query = "SELECT e.name, e.salary, d.name as department"
  + " FROM (Employee e INNER JOIN Employee m ON m.id = e.manager)"
  + " INNER JOIN Department d ON d.id = e.department"
  + " WHERE e.salary > m.salary";

Connection conn = DriverManager.getConnection(...);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
while ( rs.next() ) {
  print( rs.getString("name") );
  print( rs.getDecimal("salary") );
  print( rs.getString("department") );
}
```

CLI exploits the completeness and power of the SQL query language, and, using JDBC/ODBC, programs can connect to various different databases without significant modifications. However, embedding SQL queries as strings in program code bypasses static, compile-time type checking in modern programming languages. The complexity of database applications means that programming using the CLI is verbose and error-prone, which normally affects the programming development in the following ways:

- type safety: statically typed programming languages rely on the compiler to perform syntax and type checking in order to detect errors at compile time, but ignore the checking against those SQL queries in the form of strings, leaving SQL related errors undiscovered until runtime.
- efficiency of development: in addition to verbose, inelegant and repetitive SQL query string concatenation, the lack of robust compile-time type checking means that development of database applications typically require more modify-build-test cycles than would otherwise be necessary in order to detect and eliminate SQL errors overlooked by the compiler.
- complexity of programming: because of the impedance mismatch problem, database applications are developed based on multiple paradigms. Programmers need to be aware of the differences between these paradigms and must explicitly code to translate between them; an extra cognitive load that distracts the programmer from focusing on the logic of the program.

In past decades, a large number of approaches have been developed to simplify construction of, and to reduce errors in, data-driven applications. Because they mitigate the object-relational impedance mismatch problem, use of object relational mapping (ORM) [Amb96] frameworks like Hibernate [BK04], JPA [KS06] and Toplink [Pur06], and simpler variations such as Active Record in Ruby-on-Rails [MPY07], have become a popular approach to simplifying database driven application development. The introduction of ORM allows a programmer to create and manipulate objects in the host programming language in a manner usual for objects in that language, while having the framework, with relatively little explicit direction from the programmer, take care of the necessary operations to transfer memory objects to or from relational database tables. A fully featured ORM framework typically supports two different database interaction styles, *navigational* [Bac73] and *set-oriented* [CB74].

Navigational interaction in an ORM context means that, from an in-memory object loaded from the database, the programmer can navigate the persistent object graph that the object belongs to, while the framework, without explicit direction from the programmer, loads the required objects from the database on demand. Similarly, when an in-memory object or object graph is modified, the framework automatically takes care of updating the database accordingly, again without explicit programmer direction.

Such accesses may be lazy or eager, depending on various optimisation considerations. This style of interaction is generally suitable and efficient when exploring or updating relatively small, localised sections of the object graph.

Set-oriented interaction in an ORM context means that queries and updates can be specified in a declarative manner using an SQL like syntax. But, whereas the primitive components of SQL queries are tables, columns and attributes, the components of set-oriented ORM queries are classes, objects and fields, i.e. the programmer remains in the host language world when specifying such queries rather than having to bridge the Object-Relational language gap. Set-oriented queries are translated to SQL, shipped to the DBMS for execution and the results are transferred back and translated to the appropriate host language objects as required. They are especially suitable for obtaining objects from the persistent object graph by query, rather than by following links, and for updates and queries that involve large numbers of objects and whose results are small (e.g. calculate the sum total of sales for one month). In the latter case, it would normally be extremely inefficient to execute such operations in a navigational approach, as that would involve executing many small queries to extract large amounts of data from the database and compute the results on the application server, whereas executing it in a set-oriented style allows the DBMS to execute a single, possibly complex, optimised query and transfer only the small result back to the application server.

The term "*Navigational*" originally arose to describe the kind of access typical in databases of the network and the hierarchical models, although it has more recently been applied to one of the styles of access supported by Object Oriented Database and Semi-Structured Database models. The term "*Set Oriented*" was originally used to describe the non-navigational alternative to explicitly quantified, row-at-a-time query evaluation introduced by early versions of Sequel.

However, run time type mismatch errors between the programming language and the database schema can occur in these frameworks, and often do during program development, especially when queries incorporate run time parameters. Techniques employed in these frameworks range from the use of external XML files for mapping metadata, to runtime type reflection and to byte code compilation. These techniques often mean that typing errors, which one would normally expect to catch at compile time, are instead surviving until runtime. Thus rigorous compile-time checking is being replaced with runtime testing.

1.1.1 XML for mapping metadata

Because of its extensible, well structured, and cross platform nature, XML [Con00] is widely used for data exchange and integration of applications with heterogeneous data. ORM frameworks commonly use XML to specify object-relational mapping metadata, which is parsed and used to instruct the process of automatic data persistence and type conversion between programming languages and relational databases at runtime. Figure 1.1 contains a sample Hibernate XML mapping file, which maps class `Customer` to the table `customers`. However, these declarative, external XML files suffer from a lack of compile-time assurance of type compatibility between programming object instances and relational data. In fact, there are three types involved in the mapping process: the type of the programming object instance, the type described in the XML metadata specification file, and the type of the corresponding relational data. Any discrepancy between these types are subject to detection at runtime when the mapping is exercised.

A number of ORM frameworks [KS06] support annotations [Mic04] as an alternative approach to the mapping of metadata. This uses a special form of syntactic metadata that can be added to the program code and processed by the compiler, or that may be stored in the byte code and accessed at runtime, in the case of Java programming language. Figure 1.2 shows how to map the class `Customer` using the annotation approach. It is different from the XML mapping approach in that the use of annotation in program code can enhance compile-time type safety by allowing for compile-time validation of type compatibility between the program object instance and the metadata specification. However, it does not support compile-time type checking against the actual database schema. Errors such as mismatched database field names or types between the metadata specification and the database can still survive until runtime.

```
<hibernate-mapping>
  <class name="Customer"
         table="customers">
    <id name="id"
        column="custid">
       <generator class="sequence"/>
    </id>

    <version name="version"
             column="version"/>
    <property name="name"
              column="name"
              type="string"/>
    <property name="address"
              column="shipping_address"
              type="string"/>
    ... ...
  </class>
</hibernate-mapping>
```

Figure 1.1: XML Mapping File

```
@Entity
@Table(name="customers")
public class Customer {
    private int id;
    private String address;
    @Id
    @Column(name="custid")
    public int getId()
        {return id;}
    public void setId(int id)
        {this.id = id;}

    @Column(name="shipping_address")
    public String getAddress()
        {return address;}
    public void setAddress(String address)
        {this.address = address;}
    ... ...
}
```

Figure 1.2: Annotation Mapping

1.1.2 Runtime reflection and byte code compilation

Whether XML or annotations are used to specify the necessary mapping information, current popular frameworks for ORM process them only at runtime. This processing requires the generation of type specific proxy objects, to handle lazy loading of objects from the database, and interleaving of code, to handle transactions, security issues and exceptions. To modify the code at runtime, runtime reflection and byte code compilation is used in most popular Java language based ORM frameworks.

Runtime reflection refers to the ability to examine and modify the type of program object instances at runtime [Ibr92, LWL05], bypassing the usual accessibility controls of the programming language. This enables applications to perform operations such as accessing private fields and methods, identifying the actual types of instances, or creating instances of certain classes using their fully qualified names — a technique known as dynamic class loading [LB98]. Java language based ORM frameworks use these techniques to construct and initialise objects from relational data according to the XML mapping definition.

Byte code compilation is a method of generating new code at runtime by directly generating compiled byte code based on runtime information. It is generally used in a quite stylised way in ORM frameworks for the generation of lazy loading proxy classes, which have the same signature as their delegated entity classes and defers the initialisation of entity instances until the method invocation occurs on the instance target.

These techniques provide the low-level infrastructure used to implement a generic persistence interface (see Fig. 1.4).

However, the use of reflection and runtime byte code compilation causes a number of problems. First, it can bypass many of the typing and visibility constraints that are imposed by the base language design — making these features particularly dangerous and prone to programmer errors. Second, it postpones aspects of compilation to runtime, with the result that errors that would normally be caught at compile time can now persist until runtime.

```
Class cls = Class.forName("Foo");
Object foo = cls.newInstance();
```

Figure 1.3: Dynamic class loading (Java)

```
Session sess = sessionFactory.openSession();
Customer cust = (Customer)sess.load(Customer.class, id);
```

Figure 1.4: Session load interface (Hibernate)

1.2 Criteria of research

The problem of the object-relational impedance mismatch has been identified early in 1984 [CM84] and much research has been devoted to this problem since then. Cook and Ibrahim, in [CI05], identify specific criteria for the analysis of proposed solutions that we will refer back to in this thesis:

Typing: This problem is traditionally viewed as the key cause of the impedance mismatch. Both programming languages and relational databases support primitive types and data structures. The difficulty lies in aligning these types. This is because the types in a programming language typically do not correspond to those in a database. The differences also exists in the representation of relationships, which in a relational model are normally defined by foreign key constraints, but are expressed as references between objects in programming languages. Another consideration is the treatment of null values. Nulls in SQL are treated as unknown values and allowed to belong to primitive types (though in a complex and somewhat inconsistent manner), while in procedural programming languages, primitive types cannot be null. Relational joins can return and treat nulls as unknown values, but dereferencing a null-pointer value (the obvious corresponding elements in an object-oriented language context) throws a runtime exception.

Static typing: Static typing is a common tool used to increase reliability in both programming languages and databases, ensuring before execution time that only valid operations are performed on data. Static typing is not a property of data, thus mapping data between programming languages and databases cannot guarantee the use of data in a consistent and type-safe way. Instead, it is a property of the system that manages data in the way it is interpreted in programs and queries. That is, if type mismatches between programs and databases could be detected at compile time (i.e. statically typed), then fewer modify-build-test cycles would be required during the development and type errors could never occur at run time.

Interface style: This refers to the difference between orthogonal persistence and explicit query execution.

Orthogonal persistence is a programmatic approach to persistence, in which data is persisted into, or retrieved from, underlying databases in an implicit manner simply by accessing programming language variables. Persistence in this approach is orthogonal because the persistence behaviour is independent of the type of the value and where it was created.

Explicit query execution is another approach to persistence, in which a call-level interface is used to invoke queries in the language and style of the database, rather than that of the programming language. This kind of approach allows programmers to interact with the database server, while it typically ends up with a multiple paradigm approach to persistence.

Reuse: This is an issue relating to compositionality of operations. Parameterised and dynamic queries enable the definition of queries for multiple execution by passing different parameter values on each invocation, or the construction of queries on demand. This helps reduce the repetitivity and verbosity of program code. Modularity in queries refers to the ability to define reusable components of queries, which can be used to construct new queries. A good solution to the integration of languages and databases should support modularity, compositionality, and reuse of data-driven program structures.

Concurrency and transactions: Database operations are grouped into transactions. Multiple transactions compete simultaneously for access to shared resources. Databases guarantee that these transactions can proceed, in apparent isolation, to either be committed or completely rolled-backed or undone. When integrating programming languages and databases, the problem arises of how to expose this essential behaviour to the programming language in a manner that matches the paradigms of the host language.

Optimisation: A proper optimisation strategy may significantly reduce the running time of query execution and any successful programming language persistence solution must support common techniques in this domain. The design of the persistence interface may make various optimisation strategies possible, and make others impossible.

1.3 Summary of contributions

The contributions of this thesis are as follows.

1. Practical, compile-time type-safe object relational mapping: We propose an approach to a Hibernate-like ORM framework, which features compile-time type safety. It differs from Hibernate in that, aside from the choice of host language, we embed the mapping metadata directly in code, like Java/C# annotations in style but with the important difference that our metadata is processed at compile time, rather than run time. We introduce an additional database schema file during compile time, which, in combination with the embedded metadata, is used for type checking. Instead of using problematic runtime byte code compilation and runtime type reflection, we use program pre-processing for our ORM syntax extension, with the purpose of minimising the modifications on the standard host language, and we use bespoke ORM module generation, based on the embedded metadata, to guarantee that all the code is checked by the compiler. In addition, code encapsulation and modularity allow us to provide a concise and type-safe object-relational interface.

2. Compile-time type-safe query language. We provide an approach to implementing an embedded, object-oriented, query language, with compile-time guarantees for type safety and syntactical correctness. The basis of this approach is the introduction of *type avatars* — a new technique to leverage the host language type checking mechanism to enable type checking across the programming language/database language divide, the use of code transformation via pre-processing, and a set of techniques borrowed from the functional programming world including higher order types, modules, type inference and phantom types.

   A type avatar is a dummy data structure, modelling type constraints in queries, which allows the compiler to perform type checking on the embedded querying language. Type avatars can be used as a general approach to the type checking of embedded domain-specific language.

3. Higher-order, type-safe and protocol-safe transactions. Higher-order types in functional programming languages permits us to achieve a simpler solution than is obtainable from the current leading aspect-oriented programming approaches to transaction management, without requiring additional processing or complex configurations. We propose a transaction interface for both simple transactions and multiple transactions, the latter one invokes function execution spanning several database sources. The proposed interface includes data types for type checking, and functions for transaction invocation and commit/rollback, both of which work together to guarantee the correct ordering of transaction invocations. The use of modules makes the transaction invocation compulsory for all database related operations, thus avoiding at compile time an error that is common in the Java database programming world.

1.4 Thesis outline

The thesis is organised as follows.

- In this (the current) chapter, we introduced the motivation of the research, which can be traced back to the problem of ameliorating the impedance mismatch between relational databases and object oriented programming languages, and describe criterion of, and introduce contributions of our research.
- In Chapter 2, we review the literature on previous related work in this area.
- In Chapter 3, we motivate and propose the approach to a compile-time type-safe object-relational mapping (ORM) framework, based on the analysis of a sample of type errors overlooked by current ORM frameworks.
- In Chapter 4, we propose a type-safe, embedded domain-specific, object-oriented query language, as the replacement to a string-based call-level SQL interface, and introduce a new technique, called a *Type Avatar*, to guarantee the type safety of the language at compile time.

- In Chapter 5, we provide the design of a higher-order, type-safe transaction interface, including support for single and multiple transactions.
- In Chapter 6, we show that this approach is practical and effective by presenting a system implementing a compile-time type-safe object relational framework, called Qanat, in the OCaml programming language and using a loosely coupled SQL database. In addition, we give the formal grammar of the Qanat ORM and querying language, and discuss an interactive bookshop example, demonstrating the functionality of Qanat.
- In Chapter 7, we evaluate our research, and identify and discuss briefly a number of avenues for future research.
- In Chapter 8, we draw some final conclusions.

Chapter 2

Literature review

In this chapter, we review previous work described in the literature of compile-time type-safe data persistence, ranging from academic research to industry products.

We refer to the criteria discussed in section 1.2 in our survey, and give a concise summary at the end of this chapter by comparing the systems in tabular form.

2.1 Object-oriented databases

Object-oriented databases(OODBs) [KL89, DDB91] unify the database model with the programming data model by storing, retrieving and modifying objects in their native form, thus avoiding the object-relational impedance mismatch.

In contrast with relational databases, which view data as tables of rows and columns, OODBs provides the capability to objects that have been created using an object-oriented programming language. This differs from relational data in that these objects have structured attributes and relationships to other objects based on inheritance, aggregation and association, behaviour corresponding to a set of operations and characteristics of types such as generalisation and serialisation, and can be modified only using the methods specified by itself. Consequently, programmers who use OODBs write programs in a native object-oriented programming language (e.g. Java or C++) to do both general purpose programming and database management. This allows to retrieve and operate on data in a efficient, navigational way: objects are retrieved by following the related references without explicit loading actions; transient instances, which are created in the memory, can become persistent by building links to other, persistent instances, or by explicitly calling persistence functions.

OODBs, however, go beyond simply adding persistence to an object-oriented programming language by including the following features [ML01]:

A unified data model shared between programming language and database, avoids the object-relational impedance mismatch without requiring additional verbose work for type conversion.

Support for complex objects and relations allows an object to contain attributes that can themselves be objects or collection of objects, which is a major extension to the original concept in the relational model of only supporting atomic types.

A class hierarchy allows any subclass instance to inherit attributes and methods from its super classes.

High performance on certain tasks. Because of navigational interaction and late binding, data is normally retrieved from the database on demand, thus providing the potential for the reduction of additional I/O loading compared to typical practice in complex cases when programmers must code database interactions manually.

Reduced programming effort resulting, it is argued, from inheritance, improved possibilities for re-use and extensibility of object oriented code, and consequent increased modularity of programs.

Object-oriented databases are considered suitable for applications with complex data structures or new data types for large, unstructured objects, such as manufacturing systems, multimedia, imaging and graphics applications. However, there also exist disadvantages, which have, in practice, limited OODB as a complement to, rather than a replacement for RDBMS.

1. Since both programming languages and databases act on the same object-oriented data model, it becomes non trivial to avoid system wide recompilation of the data when an update of the schema is required.

2. A vendor specific OODB is typically tied to a specific object-oriented programming language, instead of being a RDBMS independent from programming languages.

Thus object-oriented database management system is generally a tight-coupling approach that requires various projects to use the same data model for resource sharing, and the same language for programming, rather than a loose coupling approach typical of SQL based RDBMS applications where the database has an independence from the programming language and is used as a long-term, business critical repository of data.

In addition to navigational interaction, most OODBs offer set-oriented querying in a declarative approach using an SQL like language, but based on object-oriented classes, objects and fields [ASL89]. Such a language enables programmers to retrieve objects from the persistent object graph by query, rather than by following reference links, and also supports update and query operations that involve large numbers of objects in an efficient way, avoiding the loading of all these objects from the data store and computing the result on the application server.

2.2 SQLJ

SQLJ [Ame99, WNR02], as a variant of Pro*C [AMSS05], is an embedded SQL approach, in which SQL statements are written directly inside the Java programming language. These embedded SQL statements comply with the ISO SQLJ standard, are statically predefined and can not change in real-time as the program is run, although the data transmitted by these statements can change dynamically.

SQLJ consists of a translator for embedded SQL preprocessing and a runtime library for code execution, and is smoothly integrated into the Java development environment, allowing programming with SQLJ and with other database libraries (e.g. JDBC) simultaneously. The programmer runs the translator on an SQLJ program using the SQLJ executable, which integrates code translation, compilation and customisation in a single step. The translation process replaces embedded SQL with calls to the SQLJ runtime library, which implements database operations in JDBC or other equivalent approaches. When the SQLJ application is run, the runtime library is automatically invoked to handle the SQL operations.

The SQLJ translator is designed as a preprocessor and supports the capability of checking embedded SQL syntax, verifying database operations against the database schema, and checking the compatibility of Java types with corresponding database types. Thus SQL type errors tend to be discovered during preprocessing, instead of runtime.

With the ISO SQLJ standard, an SQLJ program is preprocessed by the translator and compiled into a binary class file, for Java programming code, and serializable profile files, for embedded database operations [WNR02]. SQLJ profiles record embedded SQL as standard SQL or alternative forms (according to the customisation) and contain details about database operations, including the types, data mode and parameters used in the operation. The generation of profiles enables SQLJ applications to be portable to various relational databases, or be customised to use vendor-specific features: specific data types, syntax or optimisations.

In addition to achieving a type-safe query language, the approach of embedding SQL operations directly in Java code is much more convenient and concise than the JDBC approach, thus reducing development and maintenance costs. Figure 2.1 and 2.2 illustrate the comparison by implementing the same query in JDBC and SQLJ respectively: querying and printing the name of the employee corresponding to a particular id number.

```
String name;
int id=1000;
Connection conn = DriverManager.getConnection(...);
PreparedStatement pstmt = conn.prepareStatement(
            "select emp.name from employee as emp where emp.id = ?");
pstmt.setInt(1, id);
ResultSet rs = pstmt.executeQuery();
while( rs.next() ) {
  name = rs.getString(1);
  print( "Name is:" + name );
}
rs.close();
pstmt.close();
```

Figure 2.1: JDBC PreparedStatement Query

```
String name;
int id=1000;
#sql {select emp.name into :name from employee as emp where emp.id=:id };
print( "Name is:" + name );
```

Figure 2.2: SQLJ Select Query

The JDBC implementation (Fig 2.1) starts by initialising a prepared statement from the database connection, binding variable values to the statement, and then executing the query and iterating through the result set for the name printing, finally ends the program by closing the connection. The SQLJ implementation (Fig 2.2) embeds the SQL statement, with a preceded #sql token, directly in the code, and performs querying on a default database connection (or optionally specifying an explicit database connection). It also supports the use of Java local variables (also known as host language variables) as input or output parameters in database operations. Each host variable is preceded by a colon; this example uses Java variable `id` as input and binds the returned name to variable `name`. Because of the support for using host variables, there is no need to iterate over the result set when returning only a single row of data.

The SQLJ methodology is compile-time type safe, in contrast with the equivalent JDBC. The translator parses the SQLJ file, guarantees proper SQLJ syntax and checks for type mismatches between SQL data types and corresponding Java host variables. Online semantics checking is also optionally invoked, depending on SQLJ settings, during this phase. That is, when online checking is specified, SQLJ will connect to a specified database to verify that all database tables, columns, stored procedures, specific SQL syntax, and the type of host variables used in the application, are compatible with the database schema.

```
#sql iterator EmpIterator (int id, String name, float salary);
EmpIterator iter;
#sql iter={ select id, name, salary from employee };
while( iter.next() ) {
  printf "Id: %d Name: %s Salary %f" iter.id() iter.name() iter.salary();
}
```

Figure 2.3: SQLJ Select Query - Strongly typed result set

SQLJ uses strongly typed result sets; unlike JDBC in that the iterator of SQLJ result sets are explicitly typed. In Fig. 2.3, we predefine an iterator, `EmpIterator`, containing three explicitly typed fields. Using this typed iterator, the results of SQLJ queries are mapped to type `int`, `String`

and `float`, and also named as `id`, `name` and `salary` respectively. The correctness of this mapping is checked and guaranteed by the SQLJ translator.

Thus SQLJ provides an interface to loosely coupled relational databases that is similar in functionality to JDBC but in a more compact and readable form and with compile-time type checking. SQLJ does, however, have a number of limitations that mean it is not yet the ideal solution to the database programming language problem:

- SQLJ cannot detect all potential type errors at compile time. Because the host language, Java, lacks the capability of distinguishing null-able/not-null primitive values at compile time, SQLJ overlooks type mismatches, such as reading a null-able `int` column into a Java primitive integer variable, which can lead to runtime exceptions being triggered.
- Because SQL statements in SQLJ are predefined and embedded in the program code at compile time, the text of the queries cannot be modified at runtime. This facility is commonly used in JDBC and other frameworks to provide support for queries based on complex interactive user interfaces.
- SQLJ does little to ameliorate the object relational impedance mismatch in that queries are explicitly in SQL and refer to tables, tuples and columns, rather than host language elements, and any complex Java objects must be explicitly translated into this form to be stored to or retrieved from the database. There is no support for convenient navigational querying, lazy loading, implicit object updates or object inheritance or association mapping that is central in object relational mapping solutions.

2.3 Static SQL query checking

Gould, Su and Devanbu introduce a static SQL query analysis technique [GSD04], based on static string analysis [CMS03], finite state automata theory [HU79] and context-free language reachability problems [MR97], to verify the correctness of dynamically generated query strings. Essentially, they track the construction of SQL strings and perform type checking on each possible dynamically generated query against the SQL grammar and typing rules.

Because the approach works on compiled Java class files as a post-compilation process, this analysis technique is compatible and can be used on any JDBC application. Figure 2.4 gives an outline of the analysis process. It consists of two steps:

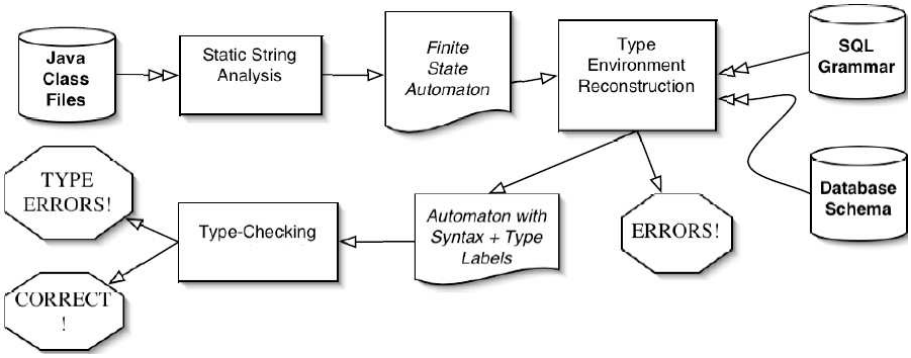


Figure 2.4: Overview of Static SQL checking [GSD04]

In the first step, the analysis builds a finite state automation (FSA), which captures a set of possible dynamically generated SQL strings for each SQL string variable that is used for query execution in the program, using the technique of static string analysis [CMS03]. The locations of the use of these string variables are called *hot spots*; i.e., locations with a call to the method `executeQuery` in the JDBC program. The string analysis starts by locating hot spots in the Java program, and then abstracts away the control flow of the program and creates a flow graph approximating the process of string manipulation for each hot spot. This yields an FSA which

represents a larger set of strings (all possible generated strings) than that actually produced by the program at runtime for each hot spot. Then the FSA is processed to get a directed graph labelled with the keywords, primitives, and literals of SQL.

Figure 2.5: An FSA with two table contexts [GSD04]

In the next step, the analysis performs semantic and type checking against the generated FSAs based on a variant of the CFL-reachability algorithm [MR97]. Because the declared types of various columns for an SQL query are only given in the database schema, rather than the FSA, it is necessary to reconstruct the typing environment and scoping information from the database schema. The indeterminacy of the type of a column makes this process non-trivial; that is, the type of any given column varies with the table it occurs in. For example, the type of column `NAME` in Fig. 2.5 can only be determined at runtime by the schema of table `TABLE1` or `TABLE2`, whichever is used for the query generation. In order to solve this problem, the CFL-reachability algorithm is applied using SQL as the context-free grammar, the obtained FSAs and the underlying database schema as inputs. It traverses each path of the automaton graph to find type context, determines tables and their column list; and, based on the schema of these tables, the type information of each column and literal is determined and annotated in the automaton graph. This is followed by a second application of CFL-reachability to perform type checking on the transformed FSA graphs. Similar to the first round, it checks the type safety of each query string represented in the FSA based on the SQL typing rules. The following is a sample of the typing rules for the integer addition and boolean value operations (and/or):

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \ + \ e_2 : int}\ \textsc{Add} \qquad \frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \ (and/or)\ e_2 : bool}\ \textsc{And/Or}$$

The above typing rule (`ADD`) reads as follows; within the typing context Γ, if both expression e_1 and e_2 are of type `int`, then the addition of these two expressions would return an integer as well. In a similar way, the other SQL typing rules can be expressed and checked.

Use of this analysis technique means that programmers could write JDBC applications as normal but have compile-time type checking on dynamically generated queries as well as on static queries. However, there are still SQL errors overlooked. The analysis does not check the type safety of query parameters and query results. In the case of separate compilation, queries constructed by string fragments from different source files can not be tracked or analysed. That is, none of the errors in the following code would be caught at compile time:

```
String name;
int id;
Connection conn = DriverManager.getConnection(...);
PreparedStatement pstmt = conn.prepareStatement(
            "select emp.name from employee as emp where emp.salary > ?");
pstmt.setString(1, "Two thousand"); //type mismatch: should be a numeric
ResultSet rs = pstmt.executeQuery();
while( rs.next() ) {
  id = rs.getInt(0); //index out of bounds
  name = rs.getString("empname"); //unknown column
  ... ...
}
```

Error 1: Type mismatch of parameters and result value. Consider the predicate condition of the query (`where emp.salary > ?`), the `salary` is compared with a numerical parameter. While the program tries to bind a string value "`Two thousand`" to the parameter. Such type errors also occur when accessing query result values, e.g. `rs.getInt("name")`, which tries to obtain the `name` of employee as an integer.

Error 2: Index out of bound. The index number of columns in JDBC result set normally starts from 1 instead of 0. Unfortunately, there is no compile-time mechanism to restrict the index number to within a correct boundary.

Error 3: Unknown column name. A quite common programmer error in database applications is to misspell the column name or use a non-existent column. When such errors occur when obtaining values from the result set, static SQL checking can not catch the error and a runtime error would be triggered.

2.4 Safe query objects

Cook and Rai introduce the concept of *safe query objects* [CR05], which allow query behaviours to be defined by using typed objects and methods in the Java programming language. In [CR06], they redefine this approach, introducing what they call a *native query API*, targeting a number of object-oriented programming languages (e.g. Java, C#).

Instead of using queries in the form of strings, safe query objects define queries as object-oriented classes. Each query class implements the safe query interface (Fig 2.6), declaring methods for filtering, ordering and existence testing. The `filter` method defines a predicate identifying whether this object meets the search criteria; if the method expression is evaluated as `true`, the item is returned, otherwise, not. The `order` method instructs the ordering direction, either ascending or descending, and criteria, on which field the ordering is performed. Because these method expressions are defined based on object-oriented classes, it is feasible to use locally defined class value in these methods and access related objects following the references in a navigational way. The `exists` method corresponds to another predicate indicating if there are items in the collection satisfying the specified query. The `execute` method defines how to invoke the query in a specified data source. In such a way, safe query objects support filtering, sorting, relationships (joins through object navigation), parameters, existential quantification, and dynamic queries, used for querying objects of a specified class. The following code [CR05] demonstrates a filtering method in safe query objects, querying all employees whose salary is more than his/her manager, which involves a join operation for comparison on employee and associated manager.

```
boolean filter(Employee emp){
  return emp.salary > emp.manager.salary;
}
```

Because queries are defined as classes and methods, potential syntax or type errors are detected by the language compiler at compile time.

```
interface ISafeQuery<T>
{
  boolean filter(T item);
  Sort order(T item);
  Collection<T> execute(javax.jdo.PersistenceManager pm);
  <R> boolean exists(Collection<R> items, ISafeQuery<R> query);
}
```

Figure 2.6: SafeQuery interface [CR05]

It is worth noting that the approach of defining query operations as native language methods for filtering items from collection data was proposed early in [CM84] and it is argued that the execution of such methods on external databases normally results in the retrieval and instantiation of all candidate objects in advance of performing in-memory execution, leading to a significant performance penalty. In order to solve this problem, safe query objects propose an optimisation solution: instead of executing these methods directly, the query class definitions are translated into code to call standard database interfaces like JDBC, JDO (Java Data Objects, a standard Java

objects persistence interface), etc, which would be invoked at runtime. This translation could be performed on the classes during compilation, on byte code after compilation, or even during load time.

The implementation of safe query objects, as an example, translates query methods into JDO operations by using the technique of compile-time template-based code translation. Specifically, it uses OpenJava [TCIK00] for compile-time meta programming for analysis and generation of code. OpenJava invokes the translation process during compile time, supplying the query class definition as input. The generated JDO code is added to the query class to override the `execute` method in the safe query base class. Safe query objects specify a query class pattern (Fig. 2.7) for identifying variables, methods and expressions in classes, and a template (Fig 2.8) for code generation. $\langle \ \ldots \ \rangle$ is used to denote repeated elements. Operator `+` corresponds to string concatenation. Each query class extends the basic `SafeQuery` class, instantiates (imported by OpenJava) `RemoteQueryJDO` indicating OpenJava to start the translation, and encapsulates a number of variables and optional methods for filtering and ordering. The translation template is defined based on the JDO API, used to rewrite the `execute` method. This method starts by initialising a query object based upon the type class on which the query would be performed, binds all the involved parameters, then sets the filtering and ordering statement using the strings translated from the method expressions, and finally executes the query and returns the result.

```
class QueryName extends SafeQuery<Type>
              instantiates RemoteQueryJDO
{
  ⟨ ParamType_i param_i; ⟩
  boolean filter(Type elem)
  {
    return filter;
  }
  Sort order(Type orderVar)
  {
    return new Sort(
      order_j, Sort.direction_j, sort_{j+1});
  }
}
```

Figure 2.7: Safe Query Pattern [CR05]

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

  Map paramMap = new HashMap();
  ⟨ paramMap.put("param_i", param_i); ⟩
  ⟨ q.declareParameters("ParamType_i param_i"); ⟩

  q.setFilter(Φ(elem, ∅, filter));
  q.setOrdering(⟨ Φ(elem, ∅, order_j) + " dir_j" ⟩);
  Object result = q.executeWithMap(paramMap);
  return (Collection<Type>) result;
}
```

Figure 2.8: Template for Generated Code [CR05]

The key point of the translation is converting the filter and sort methods into JDO query strings. Because both the JDO query language and safe query objects are designed based on object-oriented Java classes, the conversion is not as complex as we might expect without considering the object-relational mapping. Meta function Φ specifies a basic rule for converting a Java filtering expression *expr* into a Java expression, the execution of which generates an equivalent JDO filtering string.

$$\Phi : variable \times Set(variable) \times expr \rightarrow expr$$
$$\Phi(v, S, e) \rightarrow \texttt{if } e = v \texttt{ then "" else if } e \in S \texttt{ then } e \texttt{ else "}e\texttt{"}$$
$$\Phi(v, S, e.f) \rightarrow \Phi(v, S, e) + \texttt{".f"}$$

Function Φ takes three arguments:

1. a parameter of the filter method, this argument identifies the entity element (i.e. `elem` in Figure 2.7) on which the filter method occurs.
2. a set of variables (i.e. $param_i$ in Figure 2.7) that are defined in the query class.
3. the Java expression (i.e. the body of `filter` method) that is going to be translated into an expression that generates equivalent JDO strings.

The implementation of Φ includes two cases:

- *expr* is expression e that is not a field-access expression. Φ is executed as:
  - If filtering body e is the same as the filtering parameter v, return an empty string indicating that no filtering action occurs and all entities will be returned.
  - Otherwise, check if e is a variable defined in the query, if it is true (i.e. e is a variable), Φ returns the value of e as the filtering condition, otherwise, return the string representation of e as the filtering condition. Since both JDO and safe query objects are based on object-oriented classes, the string representation of e can be directly used in JDO without considering the object-relational mapping.
- *expr* is in the form of `e.f`, i.e. obtaining the value of field f from an entity expression (such as `employee` or `employee.manager`). Φ translates this expression into a string form field by invoking Φ on e in a nested manner.

Because of using JDO as the query implementation, safe query objects inherit the benefits of navigational querying; that is, programs can navigate from one instance to its related instances by following references without explicit invocations of database queries, and any modifications to persistent objects, which are loaded from and managed by the JDO persistent manager, would be reflected to the underlying database when the persistent manager is closed. However, one of the less discussed issues in this approach is the storing of object-relational mapping information. As in the Hibernate framework, JDO typically adopts XML files as the definition of the object-relational mapping metadata, thus it suffers the same type safety problem, which we have discussed in section 1.1.

Although safe query objects support filtering, sorting, parameterised existential quantification, and dynamic queries — even within the refined native query API — no solution to the support of aggregate functions and bulk data update is discussed. Without these facilities, it would normally be extremely inefficient to execute such options in a navigational approach, as that would involve executing many small queries to extract large amounts of data from the database and compute the results on the application server, or update these modifications back to the database by invoking many more queries, in the latter bulk data update. Further, because of the underlying use of JDO, safe queries currently can only return objects, instead of a collection of a number of single fields selected from the object. This further restricts the flexibility of the query mechanism.

2.5 SQL DOM

McClure and Krüger investigate another query model, SQL DOM (Database Object Model) [MK05]. It differs from Safe Query Objects in that SQL DOM generates queries through constructing a set of object-oriented classes mapping to the schema of relational databases, instead of defining a static class.

The model consists of pre-generated classes for each table and column, including classes representing statement *select*, *update*, *insert* and *delete* for each table, and an attribute class and a *where-condition* class for each column. These classes are created by `sqldomgen`, an executable distributed with SQL DOM, which inspects the relational database schema and generates corresponding classes in the form of a DLL (C# dynamic link library) file, equivalent to compiled byte code in Java. Instead of using error-prone string concatenation, dynamic queries are constructed by using instances of the generated statement, attribute and where classes, where names of tables and columns are incorporated into the class names and enumeration members, and data types of columns become types of constructor and method parameters. At runtime, the SQL query strings are generated from these constructed objects. Such an approach enables the compiler to eliminate the possibility of SQL syntax, misspelling and data type mismatch problems in the process of SQL construction.

The following code gives a snapshot of the SQL construction, obtaining the id and name of employees in a specified company, in SQL DOM. Queries are constructed based on each of the four types of SQL statement, column and where-condition classes:

```
String companyName = "..."
EmployeesTblSelectSQLStmt sql =
      new EmployeesTblSelectSQLStmt(EmployeesTblSelectColumn.EmpID,
                                    EmployeesTblSelectColumn.EmpName);
sql.AddWhereCondition(new CompanyNameWhereCond(companyName));
return sql.GetSQL();
```

Each class is associated with a single table; that is, for a specified table, the SQL DOM model generates statement classes for *select*, *insert*, *update*, and *delete*, such as `EmployeesTblSelectSQLStmt` used in the example. The constructors of each class are typed to take only parameters representing columns of the table with which the class is associated. The methods in statement classes correspond to query clauses for filtering, ordering, and relational joins; the invocation of these methods implies adding corresponding clause strings to the final generated query. In addition to the method for adding *where* condition, the *select* statement has a number of order-by methods, indicating the ordering direction on each of encapsulated column, and `JoinTo<TableName>` methods, for each table with which they share a foreign key relationship. The `JoinTo<TableName>` implicitly adds the join condition, relieving the programmer from the responsibility of having to remember the names of the foreign and primary key columns.

Each column class corresponds to one column in the relational tables and is used as a parameter to the constructors and methods of the SQL statement classes. They are used to specify which columns are to be selected, updated or inserted. Because it is quite common that some tables have columns with the same name, the column class is designed within a specified table name space.

Where-condition classes are used to specify conditions in the where clauses of select, update and delete statements. For example, we use it in the above example to set the filtering criteria on user specified company name.

Compared to safe query objects [CR05], SQL DOM is much more close to traditional SQL string concatenation. The difference is that it constructs queries by using pre-generated classes, instead of using strings. Because these classes are closely bound to the underlying database schema and type checked by the compiler, potential syntax or semantic errors in query construction tend to be detected during the compilation.

However, this approach still exhibits a number of shortcomings:

1. Complex and low-level interface. Even though the classes used in SQL DOM are correctly generated by `sqldomgen`, it is still quite complex to use this model in a database application which involves a large number of tables and columns, because, in addition to classes for where-condition, each table has at least four generated classes (select, update, insert and delete statement), and each column also has three corresponding classes for the action of select, update and insert. Using SQL DOM means using a large number of additional classes in the application, and users are still programming, in a rather verbose manner, on the database schema for querying, and on the language model for the computation.

2. Incomplete support for SQL. The SQL DOM model provides classes for database operations like *select*, *update*, *insert* and *delete*, however, it does not propose a complete structure to handle operations like *aggregate* functions, *Cartesian* products, *group* operations, and arbitrary *computation* in the *where* condition. Because the return columns can only be specified by the parameters of constructors of the *select* statement, which is generated for a specified table, it is incapable of returning columns for multiple tables which are associated by the *join* operation.

3. Type safety problem. SQL DOM focuses on the process of query construction, but overlooks the process of result set handling. At runtime, queries are returned from the query object and shipped to the database server for execution, then the result set is returned and handled using the JDBC interface, which, as we discussed in section 2.3, normally defers potential errors to runtime.

2.6 Haskell/DB

Haskell/DB, proposed by Daan Leijen and Erik Meijer [LM99a, Lei03], embeds, in Haskell, a domain-specific query language for expressing queries and other operations on relational databases in a type safe way. It embeds the terms and the type system of the SQL language into the Haskell framework, which dynamically translates and executes programs written in the embedded language, without changes to the syntax or additions to the host language. Meanwhile, the type safety of the embedded domain-specific language is guaranteed by the compiler of Haskell.

Instead of sending plain SQL strings to a database, queries are expressed with normal Haskell functions/expressions based on the relational algebra. These database expressions are checked by the compiler and translated into SQL strings, which would be executed in the database server through a low-level interface. Instead of getting an error at runtime such as the non-existence of a field name, a type error is given at compile time that points to the location where the name is misspelled. Since queries are first-class values, they can be stored in data structures, passed as arguments or take typed parameters, and used to construct complex queries through component composition. Because of using relational algebra as the intermediate representation, queries in Haskell/DB could be ported to various relational databases. Support for connection to multiple databases is added in [BHA+04].

The design of Haskell/DB consists of two parts, one for embedding SQL terms and one for type checking.

2.6.1 Term embedding

Haskell/DB uses data structures to express the abstract syntax of the SQL language, together with utility functions for a concise interface. For example, primitive expressions could be described as [LM99a]:

```
data PrimExpr = BinExpr BinOp PrimExpr PrimExpr
              | UnExpr UnOp PrimExpr
              | ConstExpr String
data BinOp = OpEq | OpAnd | OpPlus | ...

constant :: Show a => a -> PrimExpr
(.==.) :: PrimExpr -> PrimExpr -> PrimExpr
(.AND.) :: PrimExpr -> PrimExpr -> PrimExpr
```

`PrimExpr` simply defines literal constants, and unary and binary operators; `UnOp` and `BinOp` are enumerations of unary and binary operators of SQL expressions. In such a way, data structures for relational algebra operations (e.g. project, restrict, etc.) are defined. The function `constant` converts data of class `Show` (data type in class `Show` are defined with function `show` for string conversion) into `PrimExpr`; function `(.==.)` takes two primitive expressions as input and generates a new expression for the comparison operation; while function `(.AND.)` generates a logic expression. These operators are surrounded with dots in order to be distinguished from standard Haskell operators. Thus SQL expressions can be defined such as `(constant 3) .==. (constant 5)`, instead of cumbersome `BinExpr OpEq (ConstExpr (show 3)) (ConstExpr (show 5))`.

2.6.2 Type embedding

The abstract syntax ensures that expressions are syntactically correct, but cannot prevent the construction of terms such as `(constant 3) .AND. (constant 5)` that might crash the application because the operator AND can only be performed on values of boolean type. In order to achieve a type safe approach, Haskell/DB introduces an extra layer on top of the abstract syntax that serves as a type system for SQL operations, by using phantom types [LM99a, CH03, FP06], a technique that uses additional polymorphic type variables to distinguish subtypes at compile time. Primitive expressions and database operations could be redefined as:

```
data Expr a = Expr PrimExpr
constant :: Show a => a -> Expr a
(.==.) :: Eq a => Expr a -> Expr a -> Expr Bool
(.AND.) :: Expr Bool -> Expr Bool -> Expr Bool
```

Data `Expr` takes an additional type parameter `a` that is not present in the data definition and is only used for indicating type information. Function `(.==.)` is now limited to only take expressions that have the same type `a` (in addition, type `a` must be of the class Eq), and produce a boolean type expression.

In a similar way, Haskell/DB uses phantom types to prevent semantic errors such as selecting non-existent columns from a relation. Instead of using a single type parameter, the definition of the selection operator (denoted as symbol `!`) involves multiple type parameters, representing both the schema of the relation and the type of the column (or attribute).

```
data Attribute = String
data Schema = [Attribute]

data Rel r = Rel Schema
data Attr r a = Attr Attribute
(!) :: Rel r -> Attr r a -> Expr a
```

The operator (`!`) indicates the access of a specified attribute of type `a` from the relation, both of which have the same schema, and produces an expression that is of type `a`.

Haskell/DB adopts monad comprehensions [Wad90], especially the do-notation, as the front interface for expressing relational algebra, thus hiding the automatic renaming of relations and attributes, making this solution much convenient than using abstract syntax directly. The use of monads allow programmers in functional style languages to structure procedures that include sequenced operations. Thus queries in Haskell/DB could be expressed as, taking the example we discussed in previous sections, querying the name of employees who works in a specified company (Π_{name} ($\sigma_{\text{companyName}=\text{“compname”}}$ Employee)):

```
do{ emp <- table employee;
    restrict (emp!companyName .==. (constant "compname"));
    project (name = emp!name)
  }
```

Haskell/DB generates, from the query expression at runtime, concrete SQL query strings [LM99b] that would be passed to database servers via the traditional call-level interface, and constructs query results from the obtained data.

2.6.3 Analysis

```
user defined query:
  do{
    q1 <- do{ x <- table employee;
              project (email = count x email) };
    q2 <- do{ y <- table company;
              project (name = count y name) };
    return (q1, q2)
   }

user intended SQL:
  select q1, q2 from (select count(x.email) from employee as x) as q1,
                     (select count(y.name) from company as y) as q2
```

Figure 2.9: User-defined query and intended SQL

However, using do-notation to define queries can generate unexpected results. When binding multiple sub-queries in a do-notation, all of these sub-queries would be flattened into the same level during the compilation because of the associativity laws of monads [KW92]. For example, user defines a query in the form of do-notation (Figure 2.9): trying to get the count of employee's email and company's name in two sub-queries respectively. Because of the associativity laws of monads, this query is optimised and transformed into an equivalent, flattened monad expression(Figure 2.10, which invokes aggregate functions on the cartesian product of table `employee` and `company`. In this case, the transformed query will return a result unlikely to correspond to what the user intended.

```
transformed query:
do{
   x <- table employee;
   q1 <- project (email = count x email);
   r2 <- table company;
   q2 <- project (name = count y name);
   return (q1, q2)
  }

generated SQL: select count(x.email) as q1, count(y.name) as q2
                 from employee as x, company as y
```

Figure 2.10: Haskell/DB do-notation query after transformation and corresponding SQL

In contrast to approaches we have discussed in previous sections, Haskell/DB provides a functional style, type-safe interface for database programming. Even though the latest version of Haskell/DB supports queries for projection, filtering, ordering and aggregate functions, it is still difficult to incorporate general *join* and *group* operations within the current type system of Haskell. One reason that this is so is the problem of expressing queries in a relational algebra without a *join* operator. Another is that Haskell/DB guarantees the type safety of database operations but without considering foreign key constraints. It requires programmers to explicitly handle the relationships between tables and clearly know the underlying database schema. Compared to ORM frameworks, it lacks support for navigational style querying, which is typically accompanied with features like implicit loading and updating, thus Haskell/DB provides a lower level database programming interface than is available in common object relational mapping frameworks..

Haskell/DB provides valuable experience in embedding a type-safe domain-specific language in a functional programming language, such as the design of abstract syntax and the use of phantom types for type safety.

2.7 PG'OCaml

PG'OCaml [Jon07, Tei07] provides an interface to PostgreSQL databases for OCaml programs. It is different from the other OCaml database bindings (e.g. PostgreSQL-OCaml [Mot]) in that PG'OCaml uses Camlp4, a pre-processing library distributed with the standard OCaml, to extend the OCaml syntax, enabling one to directly embed SQL statements inside OCaml code. Moreover, it uses the *describe* feature of PostgreSQL to obtain type information for the embedded SQL. This allows PG'Ocaml to check at compile time the consistency between program and the underlying database schema and to generate proper code for query execution and explicit type conversion of the result data.

PG'OCaml consists of two parts, an underlying database driver, which communicates with PostgreSQL server directly by implementing the message-based frontend/backend protocol [Gro09] in pure OCaml code instead of wrapping the C libpq library, and a Camlp4 layer, which extends the OCaml syntax through pre-processing to achieve SQL embedding and type safety checking.

Specifically, Camlp4 is a pre-processing library for the OCaml language and is normally invoked during compilation. It parses the OCaml code as an abstract syntax tree (AST) and provides programmers the opportunity to perform code transformation or generation, then the transformed

AST is shipped into the compiler for ordinary compilation. Because Camlp4 is integrated in standard OCaml, this process is invoked without extra effort by the programmer.

Using Camlp4, SQL statements can be embedded in OCaml code in an extended form:

```
let get_users dbh =
      PGSQL(dbh) "SELECT id, name, age FROM users"

val get_users: PGOcaml.connection -> (int32 * string * int) list
```

It takes the form of the macro `PGSQL`, followed by the database connection handler between parentheses, an optional sequence of strings with the statement flags, and a mandatory string with the actual SQL statement. The optional flag sequence indicates whether the SQL statement should be executed at compile time, such as those (CREATE/INSERT statements) for initialising the database structure. As demonstrated in the above code, function `get_users` queries the database to retrieve user information, and returns an explicitly typed list, which highlights another feature that PG'OCaml is able to infer the correct OCaml types that correspond to the PostgreSQL types declared in the embedded statements. In particular, statements that return no data (such as the *insert/delete* statement) have type `unit`; likewise, *select* statements will typically return a list of explicitly typed tuples.

In order to verify the correctness and obtain the types of the embedded SQL, PG'OCaml makes a connection to the database at compile time by using environment variables, which inform PG'OCaml how the target database should be accessed, and sends the SQL to the database server for parsing and obtaining the types of the SQL statements, including the return value and involved parameters. Errors, found during this phase, will be reported directly. These obtained types, however, are defined by PostgreSQL and are normally inconsistent with the OCaml type system. PG'OCaml defines a type mapping between PostgreSQL and OCaml types. Consider, for example, that due to requirements of garbage collection, the `int` type in OCaml is actually 31 bits, instead of the 32 bits used in other languages and in PostgreSQL. In order to keep the full 32 bit range of SQL integer types, type `int` in PostgreSQL is mapped into `int32` in OCaml. All character types are mapped into OCaml's string. Calendar types are mapped into corresponding OCaml structures, which model the time, date, and year, etc. SQL supports declaring certain columns as NULL, these Null values in SQL are naturally captured by OCaml's optional types, in which the None has the same semantics as the NULL in SQL.

By connecting to and letting the database check the consistency of OCaml code and embedded SQL, PG'OCaml tends to detect type errors at compile time, and can support most of the SQL statements including parameterised and PostgreSQL-specific statements. However, because the embedded SQL is directly used in the database server without involving any transformation, it limits PG'OCaml to return result and accept parameters as primitive OCaml types (the one predefined in the type mapping), without using any user defined types. In addition, PG'OCaml lacks the support of constructing queries in a dynamic approach or through query composition, because each query needs to be ready and checked by the database server at compile time.

PG'OCaml is normally viewed as a pragmatic approach that relies on the database server to guarantee the type safety of SQL embedded in the program, rather than the compiler. In its current form, it has a dependency on the PostgreSQL *describe* statement, which is not supported by standard SQL. This would have to replaced by an alternative mechanism before PG'OCaml could be adapted to support other relational database management systems.

2.8 Machiavelli

Machiavelli is a polymorphic typed database programming language [OBBT89]. Its support for type inference [OB88] makes its polymorphism more general and appropriate for database applications. In particular, a function that selects the value of field f from a labelled record is polymorphic in the sense that it can be applied to any record containing field f but without constraining it to be the same type.

By employing the pre-defined primitive types (e.g. *set*, *record* and *reference type*) and database operations (e.g. *projection* and *join*), Machiavelli models relational and object-oriented databases as a set of records, thus avoiding the impedance mismatch problem naturally.

Machiavelli is designed in the spirit of the programming language ML. It inherits ML's feature of complete static type inference and polymorphism, and extends it (in the context of ML in the 1980s) to support variants, sets, general recursive types and database operations including join, projection, etc. These extensions, other than the database ones, have since been incorporated in today's ML language variants. They enable Machiavelli to define databases supporting complex structures including non-first-normal form relations, nested relations and complex objects. In particular, relational databases are represented as sets of labelled records, where each record corresponds to a row of data in the table.

Before defining database queries in Machiavelli, we give a brief introduction to types *set* and *record* and their related operations, which form the basis of the database model in the programming language. Machiavelli supports a *set* structure $\{\gamma\}$ over any data type γ for which equality is available. The Set type is defined with four basic values and functions: $\{\}$ and $\{x\}$ are constructors for the empty set and the singleton set respectively, `union` and `hom` are primitive functions for set union and homomorphic extension. `hom`, similar to `fold` in functional languages, is defined as

```
hom (f, op, z, {}) = z
hom (f, op, z {x1, x2, ..., xn}) = op(f(x1), op(f(x2), ..., op(f(xn), z)...))
```

$\{x_1, x_2, ..., x_n\}$ is the short expression of a set consisting of elements from x_1 to x_n, equivalent to $\texttt{union}(\{x_1\}, \texttt{union}(\{x_2\}, \texttt{union}(..., \{x_n\})))$ The result of `hom` is calculated by applying function `f` on each element, then the generated intermediate values are associated by function `op` which uses `z` as another input parameter; in the case of empty set, `z` is returned directly.

In addition to these primitives, a number of functions are defined using `hom` and `union`, including `map`, `filter`, and set operations like set intersection, membership in a set, set difference, the Cartesian product (prod) of sets and the power set (the set of subsets) of a set. For example, function `map` and `filter` are defined as:

```
map (f, S) = hom((fn (x) => {f(x)}), union, {}, S)
filter (p, S) = hom((fn (x) => if p(x) then {x} else {}), union, {}, S)
```

Where `(fn ... => ...)` denotes a lambda abstraction, `map(f, S)` is the set of results of applying `f` to each member of `S`, and `filter(p, S)` is the set of elements of `S` that satisfy predicate `p`.

A record type in Machiavelli can be defined as $[l_1 : \gamma_1, ..., l_n : \gamma_n]$, a collection of pairs of label and type. The record type is defined with primitive operations including `project`, `con` and `join`.

- `project`(r, σ), if `r` is value of type γ and σ corresponds to a substructure of `r`, then `project` defines the selection of substructure fields from `r`.
- `con`(r_1, r_2), determines if two records r_1 and r_2 are consistent, i.e. if they could possibly have been projected from the same record.
- `join`(r_1, r_2), concatenates two records when they are consistent, otherwise a type error is reported at compile time.

As an extension, both function `project` and `join` could be applied to *set* values. When $\{\gamma\}$ is a set type, `project`$(S, \{\gamma\})$ returns the projection on each set element. When the `join` operation is applied to two sets of records, it concatenates all the consistent records generated by the production of the two sets, equivalent to the natural join of the two relations.

```
project(S, {γ}) = map((fn(x) => project(x, γ)), S)
join(S1, S2) = map(join, filter(con, prod(S1, S2)))
```

In addition Machiavelli includes types, *reference* and *variant*, and expressions, *field-access expression* and *pattern matching*.

1. `ref(e)` represents initialising a reference that points to value `e`, `!e` represents dereferencing, i.e. getting the target value.
2. $< c_1 : \gamma_1, ..., c_n : \gamma_n >$ declares a variant type that can be constructed by using constructor c_i that has a parameter of type γ_i.

3. `r.f` denotes retrieving the value of field `f` from record `r`.

4. (case v of c_1 of x_1 =>$e_1, \ldots, c_n$ of x_n =>e_n) is a pattern match for each constructor in a variant.

Based on these types and functions, Machiavelli models relational databases as sets of records, and supports querying these set structures in the functional approach. In addition, queries in Machiavelli can be defined in a concise way using some syntactic sugar. Consider queries in the form of *select-where-from* can be mapped to equivalent expression using `map`, `filter`, and `prod`, where `E` and `P` represent the projection expression and predicate condition respectively.

```
select E where x1 <- S1, x2 <- S2, ..., xn <- Sn with P
≡ map((fn(e, p) => e),
     filter((fn (e, p) => p),
         map((fn (x1, x2, ..., xn) => (E, P)), prod(S1, S2, ..., Sn))))
```

Object-oriented databases in Machiavelli are defined as sets of record references. In order to capture class inheritance, the concept of *view* is introduced: instead of defining record types for each concrete class, a super record type contains all information in the object graph; in this way, a database consists of a set of this record references; a view represents a set of relatively simple records that contain part information of this super record, for example, a set of sub-class instances.

Suppose the object-oriented database consists of classes for `employee` and `customer`, both of which inherit from the super class `person`. Specifically, each person is registered with the name information; in addition to attribute `name`, employees have `salary`, and customers have `address`. This could be integrated in the programming language as

```
PersonObj = ref([Name: string, Salary: <None: unit, Value: int>,
                 Address: <None: unit, Value: string>])
Person = [Name: string, Obj: PersonObj]
Employee = [Name: string, Salary: int, Obj: PersonObj]
Customer = [Name: string, Address: string, Obj: PersonObj]

PersonView(S) = select [Name=(!x).Name, Obj=x] where x <- S with true
EmployeeView(S) = select [Name=(!x).Name, (Salary=(!x).Salary as Value), Obj=x]
                  where x <- S
                  with (case (!x).Salary of Value of _ => true, other
                  => false)

PersonView : {PersonObj} -> {Person}
EmployeeView: {PersonObj} -> {Employee}
```

Where `(e as l)` is a shorthand for `(case e of l of x => x, other raise Error)`. `PersonObj` is the central type of the data model, which contains all of fields in the inheritance graph, and represents `Salary` and `Address` fields as variant types, thus, in the case that the object represents an `employee`, the value of `Salary` is set, otherwise it is set as None. The `PersonView` returns all objects as plain `Person` from the database, mapping from the set of `PersonObjs` to the set of `Persons`; `EmployeeView` filters all `PersonObj` objects whose salary is not none, namely the set of all employees.

Machiavelli introduced, in database programming languages, the concept of type inference and polymorphism, which enables writing generic programs suitable for database models in a concise and type safe way. By extending the type system to incorporate primitive types (e.g. set and record) and database functions (e.g. projection, join, etc), Machiavelli provided a natural representation for both relational and object-oriented databases, and defined query operations as statically type-safe set expressions, avoiding the impedance mismatch problem.

However, because the database model is integrated in the program, the life cycle of database data is tightly coupled to that of the application. In other words, Machiavelli implements a tightly coupled persistent solution. Thus it cannot connect to a loosely coupled SQL server but controls the entire database itself.

2.9 Strong types for relational databases

Silva and Visser [SV06] present another approach to modelling relational databases in Haskell. Instead of using the *Set* type (as in Machiavelli), their approach abstracts relational databases as *Map* structures that maps primary keys to the corresponding non-primary key attributes, capturing a principle feature of relational model: the primary key identifies the relational tuple. It also differs from Haskell/DB, which uses monad and phantom types for the compile-time type safety, in that this approach uses Haskell's multiple-parameter type-class with functional dependencies [HHJW96, Jon00, Hal01] for type checking and constraints. Moreover, it allows various constraints (e.g. foreign key constraints) to be enforced on the database model.

Prior to discussing the model used in this approach, we review some aspects of the Haskell type system [SV06, KLS04], especially the multiple-parameter type class.

1. Algebraic data type

   Haskell supports algebraic data types. For example, the definition

   ```
   data Bool = True | False
   data Tree a = Leaf a | Node (Tree a) (Tree a)
   ```

   introduces the constructors `True` and `False`, the type `Bool`, and states that `True` and `False` belong to the type `Bool`. The second instance defines a polymorphic data type, where `a` is the type argument, `Leaf` and `Node` are parameterised constructors, which can be applied to arguments of an appropriate type, generating an instance of the data type to which the type constructor belongs.

   Operations on algebraic data types can be defined by using pattern matching to retrieve the arguments. For example, consider a function to find the depth of a Tree:

   ```
   depth :: Tree a -> Int
   depth (Leaf a) = 0
   depth (Node left right) = 1 + max (depth left) (depth right)
   ```

   `max` is standard Haskell function for returning the greater of two integers. `::` denotes the type of the function, given a value of `Tree` type, the function returns an integer.

2. Type classes

   Type classes introduce ad hoc polymorphism to Haskell and are normally used for defining overloaded functions. Classes are defined by stating their names and parameters and giving the types of the overloaded operations that they should support. As an example, a class for types that support showing their values as strings can be defined as:

   ```
   class Show a where show :: a -> string
     instance Show Bool where
       show True = "True"
       show False = "False"
     instance (Show a, Show b) => Show (a, b) where
       show (a, b) = "(" ++ show a ++ "," ++ show b ++ ")"
   ```

   Types are declared as belonging to a class by providing the implementation of the declared functions. The above code defines type `Bool` as an instance of class `Show`. The second instance demonstrates how classes can be used as type constraints to put a bound on the polymorphism of the type variable of the class. Given types a and b are instances of class `Show`, tuple `(a, b)` would be inferred as an instance as well.

   Type classes [Jon00] can have more than one type parameter, known as multiple-parameter classes.

```
class Convert a b | a -> b where convert :: a -> b
  instance Show a => Convert a String where
    convert = show
  instance Convert String String where
    convert = id
```

The clause `| a -> b` denotes a functional dependency [Hal01] among type parameters, which declares that type parameter a determines type parameter b.

Single-parameter type classes can be viewed as predicates on types, and multiple-parameter type classes as relations between types. In the case of multiple-parameter type classes, when a subset of the parameters functionally determines all the others, type classes can be interpreted as functions on the level of types, calculating some types from the others. Unlike functions that are defined on data values and evaluated at runtime, these type computations of multiple-parameter classes are carried out by the type checker at compile time, known as static computation [Hal01].

In addition type classes can also be used as type constraints. For instance,

```
data Show a => Tree a = Leaf a | Node (Tree a) (Tree a)
```

Here type `Tree` has been redefined with a type constraint, which limits the constructors so that they can only be applied to instances of class `Show`.

3. Heterogeneous lists

   Heterogeneous collections refers to a data type that is capable of storing data of different types, while providing operations such as look-up, update, iteration, etc. They are fundamentally different from the more usual, homogeneous collections, which hold elements of the same type only. HList [KLS04] is a Haskell library for such strongly typed heterogeneous collections, used for the representation of arbitrary-length tuples, and extensible records.

```
data HNil = HNil
data HCons e l = HCons e l

class HList l
  instance HList Nil
  instance HList l => HList (Cons e l)

tupleExm = HCons 1 (HCons True (HCons "foo" HNil))
recordExm = Record (HCons ("lab1", True) (HCons ("lab2", "foo") HNil))
```

   Data types `HNil` and `HCons` represent empty and non-empty lists, respectively. The `HList` class establishes the condition on heterogeneous lists that they must be built from successive applications of the `HCons` constructor and terminated with `HNil`. The `tupleExm` shows that elements of various types can be added to a list. Records, in a similar way, can be modelled as heterogeneous lists of pairs of labels and values. Constructor `Record` is only used to distinguish record lists from other lists.

   HList provides numerous operations on heterogeneous lists (e.g. append, insert, delete) and records (e.g. lookup value by label, modify value). These functions constitute the basis of database operations.

Employing multiple-parameter classes and heterogeneous lists (HList), Silva and Visser provide a type-safe *Map* model with database schema, attributes and foreign key constraints, to represent relational databases. For example, relational table `people`, which contains columns for id (as primary key), name and living city, could be defined as:

```
peopHeader = (attrID .*. HNil, attrName .*. attrCity .*. HNil)
peopTable = Table peopHeader
   (insert (1 .*. HNil) ("Ralf" .*. "Birmingham" .*. HNil)
   (insert (2 .*. HNil) ("Bob" .*. "London" .*. HNil) Map.empty))

data Attribute typ name

data ID; attrID = ⊥ :: Attribute Int (PEOPLE ID)
data NAME; attrName = ⊥ :: Attribute String (PEOPLE NAME)
data PEOPLE a; people = ⊥ :: PEOPLE ()
```

Each relational table in this model contains header information and a map from key values to non-key values, each with types dictated by the header. `(.*.)` is a shorthand infix operator for the concatenation operation (HCons) of heterogeneous list. The header (`peopHeader`) contains attributes for both the key values and the non-key values. Data type `Attribute` contains two type arguments, indicating the type and name of a column. Because the same column name could be used in several tables, attribute names are specified with the corresponding table name through a single-argument data constructor. For instance, the name of column `id` is defined as `PEOPLE ID`, indicating this column belongs to table `people`. As these types are only used to convey type information, they are normally defined as abstract types (without implementation, e.g. `ID`, `PEOPLE`). ⊥ is a dummy (undefined) value that could inhabit any Haskell type. In this example, it is declared with explicit type information to represent table attributes.

The well-formedness of the header and the correspondence between the header and the table values is protected by the class constraint `HeaderFor`.

```
data HeaderFor h k v => Table h k v = Table h (Map k v)

class HeaderFor h k v | h->k v
  instance(AttributesFor a k, AttributesFor b v, HAppend a b ab,
     NoRepeats ab, Ord k) => HeaderFor (a,b) k v

class AttributesFor a v | a -> v
  instance AttributesFor HNil HNil
  instance AttributesFor a v => AttributesFor (HCons (Attribute typ name) a) (HCons typ v)
```

This states that the header (`h`), key values (`k`) and non-key values (`v`) of a table satisfy the class constraint `HeaderFor`. The dependency `h->k v` indicates that the types of key and non-key values in a table are uniquely determined by its header. Moreover, `HeaderFor` enforces the constraints that attributes `a` are mapped to the key value, attributes `b` are mapped to non-key values, and there are no repeated attributes in the union of `a` and `b`. Similarly, `AttributesFor` defines the mapping from attribute collection to column values.

In addition to headers of individual tables, this model captures the relationships among tables by specifying the foreign key constraints as

```
data FK fk tbl pk = FK fk tbl pkAttribute
```

`attrCity'` and table `cities` are defined in a similar way

```
cityFK = FK (attrCity .*. HNil) cities (attrCity' .*. HNil) .*. HNil
```

Where `fk` is the list of foreign key attributes, `tbl` and `pk` are the name of the table to which the foreign key refer and the corresponding primary key attributes. For instance, `cityFK` defines the foreign key constraints from column `city` (`attrCity`) of table `people` to the primary key column `city'` (`attrCity'`) of table `cities`.

Using these tables, attributes, and foreign keys, a relational database is defined as a record, where each label is a table name and each value is a tuple of table and related constraints.

```
database = Record (cities .=. (citiTable, HNil) .*.
                   people .=. (peopTable, cityFK .*. HNil) .*. HNil)
```

For a relational table to be well formed, at the schema level all attribute names must be unique and foreign keys must refer to existing attribute names and tables, while, at the data level, values stored in a column must respect the specified constraints (e.g. referential integrity, and value ranges). Using type classes and functional dependencies, such constraints can be easily modelled by defining predicates, specifically, type classes with boolean member functions.

```
class CheckRI db where checkRI :: db -> Bool
class NoRepeatedAttrs db

data (NoRepeatedAttrs db, CheckRI db) => RDB db = RDB db
```

Data type `RDB` enforces databases to satisfy the schema-level constraints `NoRepeatedAttrs`, and run the `checkRI` predicate to check for referential integrity. Note that function `checkRI` is defined as a static computation [Hal01] that would be invoked by the type checker to iterate over each of relational tables in the database trying to find foreign key values that refer to non-existent primary key values.

Apart from the typed database model, Silva and Visser define a number of database operations for querying, including filtering, ordering, join, group and aggregate functions, and updating. Similarly, type safety is guaranteed by using multiple-parameter type classes and functional dependencies. Because the SQL language shields off the distinction between key attributes and non-key attributes, this distinction is relevant for the behaviour of constructs like *join*, *projection* and *grouping*, etc. A conversion between pairs of lists and concatenated lists is necessary and defined as

```
class Row h k v r | h-> k v r where
  row :: h -> k -> v -> r
  unrow :: h -> k -> (k, v)
```

where `h`, `k`, `v`, `r` represent table header, primary key values, non-key values and records, respectively. It is similar to the type class `CheckRI` that type class `Row` is defined on the type level and calculated at compile time. Based on this conversion, database operations, for instance, the `where` clause, can be defined as a predicate over records:

```
liveInBirmingham = λr -> (r .!. attrCity) ≡ "Birmingham"

liveInBirmingham :: (HasField (Attribute String CITY) r String) => r -> Bool
```

The type safety of these operations are guaranteed by the compiler.

The Strong Types for Relational Databases project presents another approach to integrating relational databases into the programming language Haskell, and uses multiple-parameter type classes with functional dependencies to guarantee, at compile time, the type safety of this model and related operations. A relational database is modelled as a heterogeneous collection of tables. A table, in turn, is modelled as a heterogeneous collection of attributes (the header of the table) and a Map of key values to non-key values (the rows of the table), both of which are stored in a heterogeneous collection. The use of type level programming maintains consistency between header and rows. Moreover, it enables capturing constraints, such as foreign key constraints and referential integrity, etc. Compared to the one used in Machiavelli, this approach achieves stronger type safety. However, it shares the shortcoming of Machiavelli, as well. Programmers using this approach cannot connect to external relational databases. Again, it is a tightly coupled solution.

2.10 Links

Wadler initiated the project, Links [CLWY06], aiming to provide a functional programming language for all three layers (the front web interface level, the middle, business logic level and the bottom data model level) of web application development. The principal design is that the Links

system distributes tasks among application layers, translates code into a suitable language for each layer and implements Hindley-Milner type checking [Mil78] for type consistency. It covers most aspects of web application development, including web interaction, concurrency, database querying, and XML programming. In this section, we focus on the database programming level of Links.

Links is a strict, typed functional programming language. However, side effects in Links plays a limited though important role, being used for updates to the database. Queries in Links are expressed in list comprehensions [Wad90, JW07], based on the idea that each table is viewed as a list of records, the definition of which specifies the type signature of each row. Thus, queries are defined by using constructs such as *for*, *where*, and *orderby*, corresponding to SQL's *from*, *where* and *order-by* clauses, and compiled into SQL, using the technique of rules and strategies pioneered by Kleisli [Won00] to optimise and translate the query syntax tree. That is, each rule transforms syntax tree nodes that match a particular pattern. At each node a rule application either succeeds, yielding a new subtree to replace the node, or fails, resulting in no change to the tree. Strategies are combinators for rules, used to control the order of rule application.

For the following examples we assume a table of words suitable for a dictionary application. This table contains three columns, all of type string, intended to contain the word in the dictionary, its type or part of speech (noun, verb, etc.) and its description. A query in Links to find the first 10 words, in alphabetic order, that have the given prefix, can be defined as

```
take(10, for (var w <-- words)
         where (w.word ~ /{prefix}.*/)
         orderby (w.word)
         [w]
```

Here `[w]` denotes returning `w` as the result; `<--` is the syntax shorthand for applying the function `asList`, which allows treating a relational table as a list on the type level. Links compiles such queries into an equivalent SQL statement. The match operator (`~`) is translated into the `like` expression of SQL, and the call to the `take` function in Links is compiled into `limit` and `offset` clauses in SQL. The value of the variable `prefix` will be supplied at runtime. The resulting SQL statement that is executed is:

```
SELECT w.meaning AS meaning, w.type AS type, w.word AS word
FROM wordlist AS w
WHERE w.word LIKE '{prefix}%'
ORDER BY w.word ASC LIMIT 10 OFFSET 0
```

Based on the SQL rewrite rules (Figure 2.12), SQL related expressions of Links are compiled to SQL expression. Figure 2.11 gives the grammar of the SQL-compilable subset of Links expressions. All terms are identified to α-conversion, that is to renaming of bound variables, field names and table aliases. Links allows using free variables in queries. Values of these variables will be supplied either at compile time, during query rewriting, or else at runtime. Note that in the rewrite rules given in Figure 2.12, vector notation $\tilde{v}$ represents $v_1, ..., v_n$, denoting tuples, record patterns and lists of tables; "from $\bullet$" represents the empty `from` clause; "table t with$(f_1, ..., f_n)$" is shorthand for "asList(table t where $(f_1 : A_1, ..., f_n : A_n)$ from db", where A_i is the column type declaration, and emphdb is the database handler on which the query is executed. `s[b/x]` denotes the substitution of `b` for `x` in `s`. In order to make the rewriting process easier, the non-terminal `s` in the grammar is extended to `s ::= ... | q`, where `q` represents the translated SQL query. by repeatedly employing these rewrite rules, expression `e` from Figure 2.11 can be transformed into an equivalent SQL query.

Using the technique of expression rewriting, Links can guarantee to compile a useful fragment of database operations into SQL. However, it still lacks many important features in the current version, such as the support of `aggregate` functions and `group-by` clauses. These problems are difficult, and Jones and Wadler have extended the list comprehension idea to include the `group by` expression [JW07] and support for `group-by` and `aggregate` functions may be incorporated in the next release.

Links does not provide support for dynamic (runtime) queries, where part of the query conditions are only available at runtime. This is a common requirement for complex, web-interface based, user generated querying. For instance, suppose we rewrite the dictionary query as a function `lookup` to take a condition expressed as a function (`word` $\rightarrow$ `bool`) in place of the prefix:

```
(expressions) e ::= take(n, e) | drop(n, e) | s
(simple expressions) s ::= for (pat <- s) s
                         | let x = b in s
                         | where (b) s
                         | table t with f̃
                         | [(b̃)]
(basic expressions) b ::= b1 op b2
                        | not b
                        | x
                        | lit
                        | z.f
(patterns) pat ::= z | (f̃=x̃)
(operators) op ::= like | > | = | < | <> | and | or
(literal values) lit ::= true | false | string-literal | n
(finite integers) i, m, n
(field names) f, g
(variables) x, y
(record variables) z
(table names) t
```

Figure 2.11: Grammar of an SQL-compilable subset of Links [CLWY06]

```
fun lookup (cond) {
  take(10, for (var w <-- words)
          where (cond)
          [w]
}
```

As the query translation is performed during compilation, it becomes possible to compile this into efficient SQL only once the condition is known. Instead of executing the function as a single query, Links returns the entire dictionary, then performs in-memory calculation for filtering and takes the first 10 elements of the filtered list. In order to allow efficient SQL compilation performed on such dynamic queries, it is necessary to inline calls to *lookup* with the given predicate. Unfortunately, the Links compiler does not support inlining at least in the current version, and inlining in general increases the difficulties in combination with separate compilation. Similarly, query composition, in the form of

```
let q = for ... where (...) in for (var x <- q) where (...)
```

which uses the result of query `q` to perform further querying, are typically executed in a two-step inefficient way, instead of being combined into a single query.

Links is a typed functional language. The compiler, implementing Hindley-Milner type checking [Mil78], guarantees the type safety of the program before it is invoked at runtime. However, when connecting to external relational databases, this type checking is not sufficient. An unaddressed problem in Links is that the type model of the relational database is detached from the underlying database schema: that is, there is no checking that the type of the database model as seen by the Links compiler is just what the programmer has specified, but is not checked against the actual database schema. Thus Links programmers may program with respect to a database model (table record) and queries that do not match the actual database schema they are using, but without any compile-time errors.

Compared to LINQ (to be discussed in the next section) or other ORM frameworks, Links lacks advanced features for database operations, such as flexible data mapping to relational databases (Links makes assumptions that names of table and columns are the same as the one in the schema), lazy loading, navigational query, data modification tracking.

```
TABLE
    table t with f̃
  → (a is fresh)
    select a.f̃ from t as a where true limit ∞ offset 0
LET
    let x = b in s → s[b/x]
TUPLE
    [(b̃)] → select b̃ from • where true limit ∞ offset 0
JOIN
    for ((f̃ = x̃) <-
      select c̃1 from t̃1 as ã1 where d1 limit ∞ offset 0)
        select c̃2 from t̃2 as ã2 where d2 limit ∞ offset 0
  → (ã1 and ã2 are disjoint)
    select c̃2[c̃1/x̃] from t̃1 as ã1, t̃2 as ã2
    where d1 and d2[c̃1/x̃] limit ∞ offset 0
WHERE
    where (b) (select c̃ from t̃ as ã where d limit m offset n)
  → select c̃ from t̃ as ã where d and b limit m offset n
RECORD
    for (z <- select c̃ from t̃ as ã where d limit m offset n) s
  → (x̃ not free in s)
      for ((f̃ = x̃) <- select c̃ from t̃ as ã
                where d limit m offset n) s[x̃/z.f̃]
TAKE
    take(i, select Q limit m offset n)
  → select Q limit min(m,i) offset n
DROP
    drop(i, select Q limit m offset n)
  → select Q limit m-i offset n+i
DROP∞
    drop (i, select Q limit 1 offset n)
  → select Q limit 1 offset n+i
```

Figure 2.12: Links database rewrite rules [CLWY06]

2.11 LINQ

The LINQ (Language-Integrated Query) framework [MBB06, Mic05], instead of creating a new functional database programming language, like Machiavelli, or embedding a domain specific language in functional programming languages, like Haskell/DB, addresses the impedance mismatch problem by extending the .NET languages to provide first-class support for querying and manipulating relational, object-oriented and semi-structured data. As querying becomes an integrated feature of the programming languages, LINQ allows query expressions to benefit from compile-time syntax and type checking.

The principal of LINQ is the design pattern of general purpose standard query operators for traversal, filtering, and projection. Based on this pattern, any .NET language can define a special query comprehension syntax that is subsequently compiled into these standard operators. Moreover, support for various data source types can be achieved by providing the implementation of these standard query operators. For instance, LINQ implements domain-specific APIs that work on XML (XLinq [Mic06b]) and relational data (DLinq [Mic06a]). The XML operators use an in-memory XML representation to provide XQuery-style querying. The relational operators provide an object-relational mapping (ORM) by executing queries as SQL directly in the underlying database.

LINQ is not the first attempt in .NET languages to integrate XML and relational data with an object model. Early in 2003, Microsoft research team developed an experimental language $C\omega$ [BMS05, MSB03b, MSB03a] by extending C#. $C\omega$ integrates models for XML and relational

schemas by introducing type extensions, e.g. *flattened streams*, *anonymous structs*, *discriminated unions*, and *content classes*, and query capabilities for these new types. Streams in $C\omega$ represent homogeneous ordered collections of values, aligned with `Iterator` in Java or `IEnumerator` in C#. Streams are defined as closures which encapsulate the logic for enumerating elements of collections. $C\omega$ supports explicit stream types, including `T*`, `T!`, `T?`, `T+`, representing streams of arbitrary length with elements of type `T`, streams with exactly one element, streams with either zero or one element (optional type or null-able type), and non-empty streams respectively. Anonymous struct types encapsulate heterogeneous ordered collections of values, like tuples in ML or Haskell, and are written, for example, as `struct{int i; Button}`. A value of this type contains a member `i` of type `int` and an unlabelled member of type `Button`. Discriminated union types, also named as choice types, are written as `choice{T; ...}`. As the name suggests, a value of this type may be of any type `T` listed in the declaration. Content classes are introduced to integrate XML schemas, XSD [MS01]. A content class is a normal class that has a single unlabelled type describing the content of that class. Based on these types, relational data, stored in tables as sets of tuples, can be represented as streams of anonymous structs. In addition, $C\omega$ provides generalised member access over all structural types. For instance, given a collection `buttons` of type `Button*`, `buttons.BackColor` returns the individual colours of each button in the collection; that is, the member access is implicitly lifted to the collection type. Besides generalised member access and explicit/implicit lifting, $C\omega$ also supports XPath-style filter expressions and SQL-style comprehensions.

Based on the experiences from $C\omega$ [BMS05] and other research [Mei07] in this area, LINQ introduced language extensions in $C^{\#}3.0$ [BMT07]. These included *query comprehension*, *λ-expression*, *implicitly typed local variables*, *anonymous types*, *extension methods*, *object/collection initialisers*, and *expression trees*. It is worth noting that most of these extensions take their inspiration from functional programming languages, and are essentially sophisticated syntax extensions that are compiled away using various type-directed translations into plain $C^{\#}2.0$, without modifications to the C#'s Common Language Runtime (CLR).

- Query comprehensions provide a language integrated syntax for queries in a manner similar to relational SQL, hierarchical XQuery, or list comprehensions in Haskell. Queries written in various query style are compiled into sequent query operators, which consist of standard constructs for filtering (*where*), ordering (*order-by*), and projection (*select*).
- λ-expressions are expressions of anonymous methods. They use type inference and conversions to both delegate types and expression trees.
- Implicitly typed local variables permit the type of local variables to be inferred from the expressions used to initialise them.
- Anonymous types are heterogeneous ordered collections of values, similar to anonymous types in $C\omega$ or tuples in ML and Haskell. Values of this type are typically automatically inferred and created from object initialisers.
- Object/collection initialisers allow the initialisation of objects and collections by giving pairs of field name and values in the form new T$\{f_1 = e_1; ...; f_n = e_n\}$. For example, new Button{BackColor = "white", Status = "on"} initialises a white colour button with status `on`.
- Extension methods extend existing types and constructed types with additional methods that are essentially static methods and can be invoked using instance method syntax. Extension methods are declared by specifying the modifier `this` on the first parameter of the methods. Given a method invocation of the form obj.m$(a_1, ..., a_n)$, the compiler checks to see if there is an instance method m supported by the object `obj`, otherwise, this invocation is processed as extension methods and translated into static method invocation m(obj, a_1, ..., a_n).
- Expression trees provide another internal representation of λ-expressions, instead of as code (delegates). Based on the invoked method's interface (either it requires code delegate or expression tree), the passed λ-expressions are compiled into delegates that can be invoked directly, or expression trees that can be parsed at runtime for SQL string generation or other purposes.

With the support of these extensions, a query to retrieve the names of employees whose salary is over 30000 can be written as:

```
var names = from e in employees
            where e.Salary > 30000
            orderby e.Name descending
            select e.Name
```

This would typically be compiled into the following query operator form:

```
var names = employees.Where(e => e.Salary > 30000)
            .OrderByDesending(e => e.Name)
            .Select(e => e.Name);
```

Among these language extensions, the ability to process expressions as typed data representations `Expression<T>` is critical to the implementation of DLinq and XLinq. Instead of loading all the relational data from databases and then executing in-memory queries, DLinq implements standard query operators by accepting queries as expression trees that are then translated at runtime into SQL for executing on the database. In addition, DLinq provides an object-relational mapping facility; that is, relational data can be traversed and queried in a navigational fashion, and tabular data returned from querying can be transformed into object-oriented instances, modifications of which are tracked by the DLinq framework and can be persisted into the underlying database when it is needed by invoking the submit method.

The LINQ project shares many design features with Haskell/DB [LM99a] and Links [CLWY06], as all these projects are inspired by functional programming concepts, the Haskell language and monad comprehensions [Wad90]. Differences, aside from the support for object-relational mapping, include the fact that queries in LINQ are based on a plain object model, while the design of Haskell/DB is based on a relational algebra that requires additional definitions for each relation (table) and attribute (table column). In addition, because of its approach of extending the host language, LINQ integrates querying facilities in a more natural way, rather than embedding it as a domain specific language which uses a special syntax to distinguish it from standard Haskell code. Unlike Links, which compiles query expressions into SQL at compile time, DLinq defers this translation process until runtime, enabling flexible support for dynamic querying and query composition.

Considered as an ORM framework, LINQ has a number of subtle differences from Hibernate. For tracking object modifications, LINQ can either use a data comparison approach that keeps copies of loaded objects and then compares the differences during submission, or it can listen to the invocation of each mutator method. Hibernate only supports the former approach.

Both LINQ and Hibernate support cascading actions on entities; that is, when one entity is persisted into the database, all of its referenced entities become persistent as well. Unlike LINQ, Hibernate extends this cascade functionality to cyclic entity graphs, for instance, the one shown in Figure 2.13. Entities `A`, `B`, `C` form a cyclic graph via object references. Persisting or removing any of these entities from the database would cause the storage or removal of the whole graph respectively. In such situations, Hibernate adopts the following approach: first break the circle-loop relation, then cascade the save or delete operation on each element, finally rebuild the cycle relationship in the case of the save action (no such action is required in the case of a cascading delete).

A further important difference between LINQ and Hibernate is apparent in the cascading of delete actions. LINQ relies on the database server to perform the delete action cascade, while Hibernate manages to perform delete actions on each related instance by traversing the entity graph, which requires retrieving related data from the underlying database. The approach taken in Hibernate, at the cost of some extra overhead, avoids the risk of leaving the ORM session (entity manager) in an unmanageable status. The session instance in Hibernate remains consistent with the state of loaded entities, while in LINQ the state of deleted instances can only be updated when the session is going to be closed (as reflected by the name of the delete action, `DeleteOnSubmit`). Relying on the database to perform cascading data removal puts the system at risk of failing to remove data; for example, when an entity field is defined with the cascade property '*delete-on-cascade*' indicating to delete the associated instance in a cascade manner, but the corresponding foreign key in the database has no cascade property set, deleting an entity that is referred to by

other entities fails directly since its associated entities are not removed by the database, there still exist foreign keys pointing to this entity.

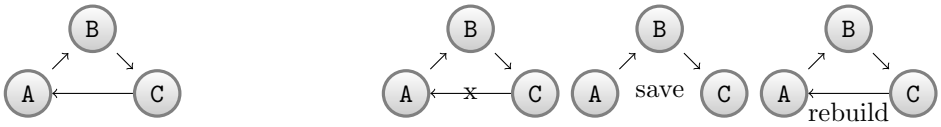


Figure 2.13: Cyclic entity graph

The LINQ framework is much more type-safe than Hibernate's HQL. Queries in LINQ are type checked by the language compiler at compile time. However, various run time type exceptions, such as null-pointer and unsupported exceptions, are possible in LINQ. Even though .NET languages introduce polymorphic nullable types for representing empty columns in specified tables, queries involving the outer join operation can return unexpected empty values, which directly throws the null-pointer exception when the result data is being hydrated. The unsupported exception is thrown in the case of trying to use, in the query, properties or methods that do not support mapping from objects to relational databases. Because LINQ allows querying against in-memory objects, the above situation is type safe but will cause runtime errors when it is used in a relational database query.

Object-relational mapping metadata is embedded as annotations in the class definitions. LINQ provides tools to inspect relational database schemas and generate class definitions with these annotations, thus making the process automatic. However, in cases where the class definitions must be modified, a not uncommon occurrence during program development, LINQ lacks compile-time checking for the modified mapping information.

2.12 PS-Algol

Unlike those approaches that involved associating persistence with the variable name or the type in the declaration, PS-Algol [ACC82, ABC$^+$83, MBC$^+$88, AM95] and variations from the same family, such as PJama [ADJ$^+$96, AJDS96, JA99] in Java attempt to provide orthogonal persistence. That is,

- Persistence for all data irrespective of their type.
- Persistence independent of how the program manipulates that data object and the state of the data object, whether persistently or transiently.
- Persistence transitivity, requiring the lifetime of all objects to be determined by their reachability.

PS-Algol implements this orthogonal persistence in the language S-Algol [Mor79a]. The concept is to identify persistence as a property that is orthogonal to data, and independent of data types and the way in which data is manipulated, so that the same code is applicable to data of any persistence. During the normal execution of a program, objects in PS-Algol becomes and remain persistent if they are reachable from the persistent system directly or from other persistent objects.

The first problem facing the design of an orthogonal persistence framework is how to handle data of various types in a uniform way. The S-Algol language itself helps the implementation of orthogonal persistence become possible. It is derived from Algol and supports representing data as an orthogonal data type *pntr* [Mor79b], i.e. a structure of a tuple of named fields that can be of various types. S-Algol runtime checking ensures that the access of field values from a *pntr* structure is performed only if the requested field is valid. In PS-Algol, persistent objects are represented as *pntr* objects, kept in the heap [ACC83] during the program execution and persisted into the backing store (either external files or database) when the program finishes. To extend runtime checking to persistent objects ensuring that the type information migrates with the object itself, type information is persisted as an implicit field of the pntr object.

```
procedure open.database(string database.name,mode,user,password -> pntr)
procedure close.database(string database.name)

procedure commit
procedure abandon

procedure lookup (string key; pntr table -> pntr)
procedure enter (string key; pntr table, value)
```

Figure 2.14: PS-Algol orthogonal persistence interface

PS-Algol is implemented as a series of functions in S-Algol while making minimal changes to S-Algol. It was not necessary to alter the compiler. These functions are designed based on the orthogonal structure and take the responsibility of moving data between the program heap and backing store:

- Data is moved to the heap when dereferencing a persistent identifier that is returned from the data retrieval function. This dereferencing action calls the persistent manager to locate the object and place it on the heap. More specifically, the retrieved object contains subsequent pointers referring to associated data objects, thus objects can be accessed in a navigational way.
- Data is persisted into the backing store from the heap when a transaction is committed. At this time all data on the heap that is reachable from the persistent manager are written back to the store.

PS-Algol persistence mainly includes functions for database opening and closing, transaction functions, and data persistence and retrieval. The interface of these functions [ACC82] are shown in Figure 2.14. Note that S-Algol defines functions as procedures: (`->`) indicates the return type of the function; symbol (.) can be used in the name of variables or functions.

1. Database open and close

   The procedure `open.database` opens a database connection using the specified database name, mode (read or write), user name and password, and returns a pointer to a table that is an index from strings to values of type *pntr*. This pointer acts as the root (origin) of persistence from which persistent data can be reached. A transaction is opened with the database connection. Procedure `close.database` closes the specified database making it available for other users who wish to write to it. PS-Algol doesn't allow multiple write-mode connections to one database at the same time.

2. Transaction commit and rollback

   The second group of procedures procedures commit and rollback the changes made since the last commit, respectively.

3. Data persistence and retrieval

   The last group of procedures provide the facility of persistence management: associative lookup of values from, and writing values into, a table that is constructed to store persistent data objects as pairs of string and pntr values.

Using orthogonal type *pntr* as the unified data representation, PS-Algol provides orthogonal persistence to data objects of various types without requiring any specific instructions, such as SQL strings written in JDBC or customised mapping configuration in ORM frameworks. PS-Algol abstracts away the persistence detail commonly visible to the programmer, lets persistence be an intrinsic property of the programming environment, thus reducing the effort required by the programmer to accomplish those tasks. It apparently results in programs that are simpler to understand and to transport.

PS-Algol persists *pntr* type data with implicit fields containing the type information that migrate with the data itself. This ensures that type checking is still available when accessing field values from persistent data objects. However, this checking is performed during the program execution, rather than at the compile time. If any type mismatch occurs between the program and the persistent data, errors are discovered only at the run time.

Even though the PS-Algol approach uses external files or databases as the backing store, the persistent data is tightly coupled with the program. Modifications on the program model, as subtle as renaming a field in data objects, may cause the existing persistent data to become invalid. The fact is that such changes occur frequently during the application development. This limitation restricts the use of the same persistent data among various applications. Further, the data model employed is specific to PS-Algol. While PS-Algol can store its data in a relational database, it cannot interact with arbitrary tables in that database — only with the data that directly corresponds to its persistent object formats.

2.13 Summary

In this chapter we discussed some of the major contributions to the object relational impedance mismatch problem and compile-time type-safe approaches to solving it.

We summarise, in Figure 2.15, the results of a comparison of these approaches based on the criteria described in Chapter 1.

	JDBC	Hibernate	OODBC	SQLJ	Static checking	Native query	SQL DOM
Typing							
Mapping	N	Y	Y	N	N	Y	Y
Nulls	N	N	N	N	N	N	N
Statically typed	N	N	N	P	P	P	P
Interface							
Orthog. Persistence	N	P	P	N	N	N	N
Explicit query exec.	Y	Y	Y	Y	Y	Y	Y
Optimisation							
Caching	N	Y	Y	N	N	N	N
Query shipping	Y	Y	Y	Y	Y	Y	Y
Bulk data manipulate	Y	Y	Y	Y	Y	N	Y
Reuse for explicit queries							
Parametrised queries	Y	Y	Y	Y	P	Y	Y
Dynamic queries	Y	Y	Y	N	Y	P	P
Query composition	Y	N	N	N	N	N	N
Transaction							
Transaction	Y	Y	Y	Y	Y	N	N

Y = Feature supported
P = Partially supported or supported with minor problems
N = Not supported

	HaskellDB	PGOCaml	Machiavelli	Strong types RD	Links	LINQ	PS-Algol
Typing							
Mapping	N	N	Y	Y	P	Y	Y
Nulls	Y	Y	Y	Y	Y	Y	Y
Statically typed	P	Y	Y	Y	P	P	N
Interface							
Orthog. Persistence	N	P	N	N	N	P	Y
Explicit query exec.	Y	Y	Y	Y	Y	Y	N
Optimisation							
Caching	N	N	N	N	N	Y	N
Query shipping	Y	Y	N	N	Y	Y	N
Bulk data manipulate	Y	N	N	N	Y	N	N
Reuse for explicit queries							
Parametrised queries	Y	N	Y	Y	Y	Y	N
Dynamic queries	N	N	N	N	P	Y	N
Query composition	y	N	N	N	P	Y	N
Transaction							
Transaction	N	Y	N	N	Y	Y	Y

Y = Feature supported
P = Partially supported or supported with minor problems
N = Not supported

Figure 2.15: Summary of evaluation of solutions to impedance mismatch

Chapter 3

Compile-time type-safe ORM

There are commercial and open-source production quality ORM frameworks available and in common use. Hence the technology for such systems is already available. The main contributions of our work is presented in Chapter 4, where we show how we embed a domain-specific, compile-time type-safe object-oriented querying language in OCaml, and in Chapter 5, where we present a higher-order compile-time protocol-safe and type-safe pattern for transaction handling. Nonetheless, we still need:

1. to explain the basic, and well understood, functional elements of ORM systems in general so that these later chapters can be understood, and

2. to explain how we overcome the engineering problems of implementing this basic ORM infrastructure in OCaml, in such a way as to support our goal of compile-time type safety guarantees.

This latter task has not been addressed in the literature before and is not entirely trivial, given the type restrictions necessary and the fact that existing ORM frameworks have been content to rely on run-time typing and non-strict typing features. We address both these tasks in this chapter.

Because they mitigate the object-relational impedance mismatch problem [Amb03b, CM84], the use of object relational mapping (ORM) frameworks like Hibernate [BK04], JPA [KS06] and Toplink [Pur06], and simpler variations such as Active Record in Ruby-on-Rails [MPY07], have become a popular approach to simplifying database driven application development. However, these frameworks have not been particularly concerned with compile-time type safety. Type mismatch errors between the programming language and the database schema occur quite often during program development, especially when queries incorporate run time parameters. The techniques used in these frameworks often defer error checking on types until runtime, as we have discussed in Chapter 1.

If such errors were caught at compile time instead, then fewer modify-build-test cycles would be required to develop the program and this kind of error could never occur at run time, thus both speeding up program development and reducing errors in deployed systems. We use designs similar to those used by Hibernate in order to achieve core functionality, but adapted, where necessary, to avoid non-strict or non-compile-time typing mechanisms:

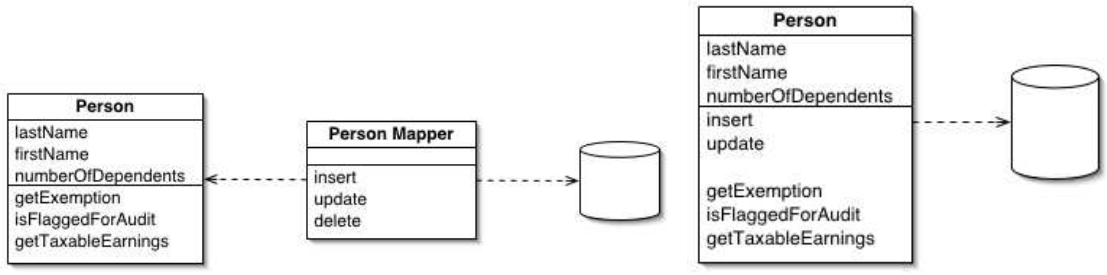
- Proxies to support lazy loading of objects from the database on demand, section 3.3.
- Session objects, obtained from session factories, encapsulating low level database connections and containing caches of original versions of loaded objects plus copies of those objects that are returned to the programmer so that, at the end of a session, the original and the modified versions of each object can be compared to determine the updates required in the database, section 3.4.
- Mapping of object references in in-memory objects to foreign keys in the database, section 3.1.
- Overloading of collection classes to manage lazy loading of 1-to-N relationships, section 3.3.2.

Where our system, which we call *Qanat*, differs from Hibernate, aside from in the host language, is in achieving compile-time type safety. Hibernate uses dynamic loading, run time reflection and byte code compilation in its implementation, all of which make compile-time type safety problematic to achieve. Our approach leverages OCaml's type inference facilities through the use of a new mechanism we call *Type Avatars*, and program transformation using the preprocessing technique, to add compile-time type safe language extension support for querying and object relational mapping. We further employ higher order functions and exceptions to cleanly handle transactions.

3.1 Object-relational mapping

In the following, we first consider the mapping strategy, then proceed to issues of type safety, and then to other ORM-related problems.

Active Record and *Data Mapper*, among various other mapping strategies and design patterns [Fow02], are employed by a large number of ORM frameworks, including Ruby-on-Rails and Hibernate.



(a) Data Mapper (b) Active Record

Figure 3.1: Design Pattern of Data Mapper and Active Record [Fow02]

Active Record: This wraps each row of data in relational tables or views as an instance of an object-oriented class that encapsulates methods for database access and domain logic [Fow02]. As shown in Figure 3.1b, class `person` internally knows how to interact with the database, providing the basic CRUD operations (`create`, `retrieve`, `update`, and `delete`), and constructors for object initialisation.

Relational tables in Active Record are directly mapped to individual objects. Because of its simplicity, active record is normally used in CRUD applications with relatively simple domain logic, especially in combination with runtime reflection and meta-programming, such as in Ruby-on-Rails, which makes the process automatic to inspect database schemas and generate mapping classes from the underlying databases. One of the negative aspects of active record is the high degree of coupling between the application model and the database structure.

Data Mapper: This is a layer of *mapper* classes that move data between the programming language and a database while keeping them independent of each other [Fow02] (c.f. Figure 3.1a). Mappers take the responsibility for data conversion. With this additional data mapper layer, in-memory objects can be unaware of even the existence of the database. For programmers, there is usually no need to write explicit SQL to perform data access, and no knowledge required of the database schema, once the data mappers have been provided.

The Data Mapper pattern provides a decoupled, customised object-relational mapping. Since the additional data layer is transparent, it lends itself to adding advanced features, such as dirty data checking, without requiring extra application programmer effort.

Following Hibernate and LINQ, our approach adopts the data mapper as the mapping strategy, implementing a session container, analogous to Hibernate's *Session* and LINQ's *DataContext*. As we use the same basic approach, we face the same ORM-related problems as well:

- Lazy loading — We need to support in-memory navigational queries that allow programmers to navigate through the object graph by following object references. This requires loading related objects as well as objects that are explicitly required, so that, when we navigate to an object, all the data for that object is available in memory as required. However, if all relationships were chased and loaded eagerly, rather than lazily, loading an entity could have the effect of loading a huge graph of related instances. In the worst case, it might load all the data in the database.

- $N+1$ fetch problem — This problem occurs when programmers retrieve an object containing a collection of N sub-objects from the database via the ORM framework, and then iterate the collection to access entities related to individual collection members. If no hint is given to the ORM system, it can handle these requests as unrelated, serial requests for independent objects, executing $N + 1$ select queries on the database. Such an approach, however, is far from optimal as the same result could be achieved by invoking a single *join* query. The $N+1$ problem typically accompanies frequent database hits that seriously reduces the performance.

- Referential integrity problem — Relational databases enforce referential integrity by using both primary and foreign keys to ensure that any field in a table that is declared as a foreign key can only contain values from the referred table's primary or candidate key columns. Thus, deleting a record that is referred to by a foreign key would break referential integrity. The design of an ORM framework faces the problem of how to enforce the referential integrity on the host programming language side of the system, especially in the case of cascading actions. When an entity is saved into the database, all of its related entities become persistent as well, however, the order of the persistence must obey the underlying database referential integrity. This means that the entities must be saved in a specific order to avoid a referential integrity error being triggered by an intermediate state in the persistence process.

- Dirty data tracking — ORM framework typically provides the facility of dirty data checking. At a point in time no later than when the database connection is going to be closed, all modifications on the persistent objects are detected and persisted into the database automatically. Various ORM frameworks take different approaches to handling this problem, either using the Listener pattern, that monitors the invocation of any modification function, or by caching data copies for dirty data checking, or by providing support for both approaches (e.g. LINQ).

In addition to these problems, type safety is another important aspect in object-relational mapping. Programming language compilers can guarantee the type safety of the part of the program that is purely written in the host language, but it is still difficult to detect all type errors at compile time in the case of connecting to a loosely coupled database. Because of the impedance mismatch and separation between programs and databases, the host language compiler lacks the capability for full type checking across host language program, database language and database schema. ORM frameworks typically use external files, such as literal XML files, for the mapping metadata, that are loaded only at runtime. Hence, any mismatch between the program, XML file and underlying database would result in a runtime exception rather than a compile-time error. Annotations [Mic04, Hib07], a language feature that enables adding the kind of information stored in these XML files as code in the host language, goes some way to ameliorate this problem — although the annotation content is often still inspected mainly at runtime — but techniques (e.g. runtime reflection, dynamic class loading and byte code compilation) employed in ORM frameworks transform the behaviour of the program at runtime [Ibr92, FF04, LB98, Dah99], thus deferring discovery of some type errors until the code is executed [Sar97].

3.2 Type safe mapping

ORM frameworks rely on the mapping metadata between program and relational database to provide object relational mapping facilities. This metadata defines the type mapping from program languages to databases, and data mapping from program variables to database columns, and relation mapping from program data associations to database foreign keys, and can be stored in external files, embedded as annotations in class declarations, or reflected from the class declaration

at runtime. Most ORM frameworks, however, only deal with these data at runtime, thus precluding compile-time type checking across the union of the database and program types. We use a number of techniques to process these metadata at compile time and reduce problems caused by changes to the database schema after compilation.

3.2.1 Code-embedded mapping metadata

Our approach is to embed the metadata directly in the code, like Java/C# annotations [Mic04, Sun04] in style but with the important difference that our metadata is processed at compile time, rather than run time. We also adopt a lesson from Ruby-on-Rails, providing default mappings for the usual cases while supporting customised mapping.

Annotations in the Java language are a special form of syntactic metadata embedded in Java source code, and can be employed for class, method, variable, parameter and package [Mic04]. They are different from Javadoc tags in that annotations can be reflective. This means that they can be embedded in class files (byte code) generated by the compiler and may be retained by the Java virtual machine and made available for inspection at runtime. Essentially, annotations themselves can not transform the behaviour of program. Rather, they can be extracted at compile-time, load-time or runtime by Java VM or third party libraries to instruct additional operations. Annotations are often used by frameworks as a way of conveniently encoding information for use by user-defined classes and methods that otherwise would have to be declared either in external sources like XML file or programmatically. ORM frameworks support declaring object-relational mapping metadata as annotations that are retained in the compiled byte code and reflected at runtime for instructing the ORM behaviour. However, these annotations can only be checked syntactically at compile time, but not semantically. This means that the validation of mapping annotations are deferred until they are actually used at runtime.

Like annotations, ORM metadata, in our approach, are embedded in the source code via syntactic extensions. However, the semantics of the metadata is checked against additionally loaded database schema information at compile time (see section 3.2.2). The language syntax is extended by employing a pre-processing technique. During this process, the mapping metadata is extracted, which then, in combination with the additionally loaded database schema information, is used to generate ORM support modules. Unlike Java/C# annotations, the embedded mapping metadata in our syntax extension is removed from the compiled byte code. Specifically, OCaml, the host language of our research, provides a built-in pre-processing and syntax extension library, CamlP4 [Jam05, dR03], which is invoked at compile time for syntax extension, code transformation and generation, and then passes the transformed code directly into the compiler. The source code written in the extended syntax (i.e. with embedded mapping metadata) is transformed into standard OCaml code, including class definitions for persistent objects and extra modules to supply the ORM functionality.

The Qanat syntactic extension of ORM class definitions minimises modifications on standard OCaml class types. We show a brief overview of this extension in Figure 3.2. The formal grammar is defined in section 6.3. Symbol $\bar{a}$ stands for a list of a elements. Italic font denotes non-terminal tokens. Two extended structures are introduced: ORM modules and ORM classes. The keyword 'orm' on the definition indicates that the structure will be preprocessed to incorporate the facility of object-relational mapping. The orm module is transformed into an OCaml module of the same name, providing ORM facilities (including save, update, delete and load) for encapsulated orm classes. An orm class corresponds to a standard class, while the embedded mapping metadata is used for the generation of the orm module and removed after preprocessing.

Specifically, orm class declarations contain a number of field expressions, each of which specifies the field information, including type, mapping column (or columns) and special properties (e.g. key generator and cascade property). Among these fields, it is necessary to define a primary-key field, annotated by an asterisk and a key generator property. These fields can be specified as *primitive* (a simple primitive field value), *optional* (the value can be null), *composite* (e.g. tuple, object, record or variant), or *associated* (fields that reference the objects of the same or different orm class). Fields identified as associated need to be defined with the corresponding foreign key and cascade property. The foreign key is predefined as meta data, *key* and *bind* (refers to *meta-key* and *meta-bind* respectively); *key* indicates the reference relationship from a column in one table

orm-module ::=
  `module orm` *modulename* `= struct` $\overline{\textit{meta-key}}$ $\overline{\textit{meta-bind}}$ $\overline{\textit{orm-class}}$ `end`

orm-class ::=
  `class orm` *clazname* `: "`*tblname*`" = object` $\overline{\textit{field-expr}}$ `end`

meta-key ::=
  `meta` *keyname* `= key {"`*tblname.colname*`", "`*tblname.colname*`"}`

meta-bind ::=
  `meta` *bindname* `= bind {`*keyname*`,` *keyname*`}`

field-expr ::=
  `val mutable *`*fieldname col-mapping* `:` *field-type* `{`*key-gen*`}`
 `|val mutable` *fieldname col-mapping* `:` *field-type*
 `|val mutable` *fieldname col-mapping* `:` *field-type* `=` *value-expr*
 `|val mutable` *fieldname* `:` *associate-type* `with` *key-bind-name* `on` $\overline{\textit{cascade-property}}$

col-mapping ::= `"`*colname*`"` | `"`$\overline{\textit{colname},}$`"`

field-type ::=
  τ | τ `option` | *field-type* `*` *field-type*
 |`<`$\overline{\textit{fieldname col-mapping} : \textit{field-type};}$`>`
 |*rname*`.{`$\overline{\textit{fieldname col-mapping} : \textit{field-type};}$`}`
 |*vname*`.[` | *vconst col-mapping* of *field-type* when “*colname*”.*val* = *value-expr*`]`

τ ::= `primitive-type`

value-expr ::= `OCaml-value-expression`

associate-type ::=
  clazname | *clazname* `option` | *clazname* `resizeSet`
 |*clazname* `resizeArray {idx : "`*colname*`"}`

key-gen ::= `auto | assign | refer_to` $\overline{\textit{fieldname},}$

cascade-property ::= `all | save_update | delete | delete_orphan | eager_fetch`

modulename, *vconst* ::= `uidentifier`

keyname, *bindname*, *key-bind-name*, *clazname*,
tblname, *colname*, *fieldname*, *rname*, *vname* ::= `identifier`

Figure 3.2: Grammar of ORM syntax extension

to the primary-key column of another table; *bind* creates a pair of foreign keys, which is used to represent the many-to-many relationship that is defined by two foreign keys existing in a third table. The cascade property specifies which kind of operation (e.g. save, delete, update) on the entity should be automatically applied to this field.

```
(* Standard OCaml class type *)

class type student =
object
  val mutable id : int option
  val mutable name : string * string
  val email : string

  (* we omit method definitions *)
end
```

```
(* Extended ORM class *)

class orm student : "students" =
object
  val mutable *id "studentid" : int {auto}
  val mutable name "firstname, lastname"
                  : string * string
  val email : string
  (* user-defined methods *)
end
```

Figure 3.3: Comparison of a standard and corresponding Qanat ORM Class

Figure 3.3 gives a simple comparison of the extension without involving class relationships. Compared to the standard class declaration, an `orm` class declaration has additional information wrapped inside double quotes, indicating the corresponding database tables or columns used for persistent storage. In the case of the `email` field, which omits the explicit column name specification, the field will be mapped to a column of the same name. Such default mapping is also suitable for the metadata (table mapping) of ORM class. Thus, by adding the keyword `orm` and marking the primary-key field, a standard class type definition with primitive type fields can be transformed into a persistent class with default ORM functionality. The asterisk in the `id` field indicates that this field corresponds to the primary key column, the value of which will be automatically generated when persisted to the database (indicated by the `auto` property). Field `name` is defined as a tuple structure, mapping to the `firstname` and `lastname` column. User defined methods remain the same as in the standard class.

For completeness, Qanat ORM extensions supports various standard and nested OCaml types, including *primitive types*, *calendar types*, *tuple*, *record*, *objects* and *variant*. It includes a new *Numeric* type, mainly to handle currency values but also to unify treatment of various numeric types. It supports single and multiple-column primary-key fields with various key generation strategies: `auto`, `assign`, and `refer_to`. Strategy `auto` indicates that the value of primary key should be automatically generated when the entity is persisted into the database, `assign` requires the user to specify the primary key value when the entity is initialised, and `refer_to` requires that the primary key of the current entity remains the same as the primary key of an associated entity (or a number of associated entities in the case the current entity's primary key consists of primary keys from multiple associated entities) that is explicitly given in the `refer_to` property.

ORM classes map database relationships as class associations (references). ORM Frameworks typically require users to annotate the foreign key information on each associated field. This means that, in the case of building bidirectional references, duplicated data (foreign key detail) is involved. In order to capture the feature that foreign key constraints in relational databases are declared once and naturally bidirectional, foreign key constraints in the Qanat framework are declared separately from the ORM classes (*meta-key* and *meta-bind* in the grammar). Although declared only once in the code, this metadata may be used multiple times in the orm class. We give full examples in Chapter 6.

3.2.2 Database schema for compilation

Qanat syntax extensions guarantees the syntactic correctness of the embedded mapping metadata. However, it is also our aim to catch type errors between the program being compiled, the mapping metadata and the actual database schema. In order to catch these errors, we use an additional database schema file extracted from the relational database. We use the following approach to

enable the OCaml compiler to check for such type errors without having to write our own type checker: During compilation, the schema file is loaded, and the orm classes with embedded orm extensions are translated into standard classes and separate ORM module that provide object-relational mapping facilities for these classes. Each orm class is translated into a standard OCaml class with optional or labeled constructor arguments corresponding to the class fields. The constructor arguments are of the types obtained, based on the mapping metadata, from the database via the schema file. The fields of the classes are declared with the types extracted from the original orm definition. If any mapping column does not exist, or if the type information does not match, a fatal compile-time type error will result. Figure 3.4 shows an example of the translated orm class.

```
class student ?(id:int option)
            ~(name: string * string)
            ~(email: string) () =
object
  val mutable id : int option = id
  val mutable name : string * string = name
  val email : string = email

  method id = id
  method set_id id' = id <- id'
  method name = name
  method set_name name' = name <- name'
  method email = email

  (* user-defined methods *)
end
```

Figure 3.4: Standard Class Translated from ORM Class Definition in Figure 3.3

The translation of orm class fields into optional or labelled arguments and standard field expression can be performed using the rules in Figure 3.5, where `argument` and `field-expr` are the generated class argument and field expression respectively, and `action` are the additional operations invoked for consistency checking during the translation. We make use of the following data and functions;

- The *metadata* variants, *Key* and *Bind*, hold a foreign key constraint in the database, which is internally designed as a record and a pair of records respectively:

  ```
  type key = {from_table: string; from_column: string;
              to_table: string; to_column: string}

  type metadata = Key of key | Bind of key * key
  ```

  Type *key* has fields for tables and columns, representing the foreign key constraint from the column in one table referring to column in another (or same) table. *bind* is a pair of keys, used to represent the many-to-many relationship, which requires two foreign keys both maintained in the third table.

- σ : *string* $\rightarrow$ *string* $\rightarrow$ τ infers the programming type of *table*'s column from the database schema, used as: σ *tblname colname*

- $\rho : \tau \rightarrow$ *string* returns the name of the primary-key field from a class, ρ *claz*.

- π : *string* $\rightarrow$ *string* obtains the mapping table of the specified class: π *clazname* returns the corresponding table name.

- κ : *key* $\rightarrow$ *key* $\rightarrow$ *string* $\rightarrow$ *string* $\rightarrow$ `unit` checks if these two keys represent a many-to-many relationship between the two specified tables, otherwise, reports errors by throwing

an exception. Suppose $table_1$ and $table_2$ has a many-to-many relationship, which is kept in a third table storing foreign key constraints key_1 and key_2, thus κ key_1 key_2 $table_1$ $table_2$ returns `unit`, passing the check.

- *get_type* : *string* → *string* → τ looks up the type of a field in one class,
  get_type clazname fieldname
- *get_metadata* : *string* → *metadata* retrieves a *metadata* variant (*key* or *bind*) according to the identifier.
- *is_primary_key* : *string* → *string list* → `unit` checks if the primary key of the specified table consists of these columns, otherwise reports a compile-time error.
  is_primary_key tblname colname_list.
- *is_optional* : τ → `bool` checks if the specified type is an optional type.
- *eq* : τ → τ → `unit` checks the consistency of two types. Expression *eq* $type_1$ $type_2$ returns *unit* if $type_1$ and $type_2$ have the same structure, otherwise an exception is thrown. It is used to check the consistency of record types and variant types.
- *exists* : *string* → *string* → *key* → `unit` checks whether a foreign key is bound to the specified two tables: *exists table1 table2 key*, otherwise an exception is thrown.

Note that, in Figure 3.5, arguments marked by a question mark are optional, those preceded by a tilde are labeled (or named) arguments. Some translations (e.g.*primary-key field*, *associated reference field*) require additional actions checking the key (either primary key or foreign key) against the database schema. Specifically, fields which are of option type or defined with a default value are transformed into optional arguments; in the latter case, when there is no value passed for this argument, the default value is used. The primary-key field is transformed according to the key generation strategy: `auto` corresponds to the optional argument, instructing the framework to generate the value when it is not specified by the user; `assign` corresponds to the labeled argument, requiring users to pass the value during initialisation. In the case of `refer_to`, because the value of the primary-key field depends on another entity, we omit the corresponding argument from the class constructor and the `setter` function, avoiding that the value is modified by mistake. Other fields are transformed into labeled arguments. Key checking performs checks against the database schema; namely, whether the foreign key (or primary key) for the specified tables exists in the database, and whether it is used in the right place to represent a single reference, optional reference or collection reference.

Because there is no direct type mapping between the host programming language and the relational database, the implementation of function σ, which infers the programming type of column from the database, requires conversion between two type systems. Figure 3.6 proposes a type conversion. Most mappings are trivial, for instance, the type of a non-nullable column that is defiend as integer in the database is mapped to OCaml's primitive `int`, while a nullable integer column is mapped to OCaml's `int option`. However, relational databases have numeric types with and without precision and scale. To build the mapping, we define a module, named `Numeric`, which includes a variant type t and operations like `plus`, `minus` `mult`, etc. Where `NumPS` maps to the numeric value with explicit precision and scale; `Num` uses the default OCaml `num` type to represent the numeric value without explicit precision and scale in databases. Function `plus` uses pattern matching to analyse the value and perform the operation.

```
Suppose the transformation is based on the orm class,
                        class orm claz : "tbl" = object ... end

(* primary-key field with various generator *)
val mutable *f "col" : τ {auto} ⇝
   argument: ?(f:(σ tbl col))
   field-expr: val mutable f : τ option = f
   action: is_primary_key tbl [col]

val mutable *f "col" : τ {assign} ⇝
   argument: ~(f:(σ tbl col))
   field-expr: val mutable f : τ = f
   action: is_primary_key tbl [col]

val mutable *f "col_1,...,col_n" :
       τ_1 × ... × τ_n {refer_to x_1,...x_n} ⇝
   argument: no argument in class definition
   field-expr: val mutable f : τ_1 × ... × τ_n =
      (x_1#(ρ (get_type claz x_1))) × ... ×
      (x_n#(ρ (get_type claz x_n)))
   action: is_primary_key tbl [col_1;...;col_n]
         no setter function generated

(* primitive type *)
val mutable f "col" : τ ⇝
   argument: ~(f : (σ tbl col))
   field-expr: val mutable f : τ = f
   action: no additional action

(* field with default value *)
val mutable f "col" : τ = v ⇝
   argument: ?(f:(σ tbl col) = v)
   field-expr: val mutable f : τ = f
   action: no additional action

(* option type *)
val mutable f "col" : τ option ⇝
   argument: ?(f :(σ tbl col))
   field-expr: val mutable f : τ option = f
   action: no additional action

(* record *)
val mutable f : r.{f_1 "c_1":τ_1; ...;
                   f_n "c_n":τ_n} ⇝
   argument: ~(f : r)
   field-expr: val mutable f : r = f
   action: eq r {f_1:(σ tbl c_1); ...; f_n:(σ tbl c_n)}

(* variant *)
val mutable f : v.[C_1 "c_11,...c_1i" of τ_11 × ... × τ_1i when "col".val = v_1 | ...
                  |C_n "c_n1,...c_nj" of τ_n1 × ... × τ_nj when "col".val = v_n] ⇝
  argument: ~(f : v)
  field-expr: val mutable f : v = f
  action: eq v [C_1 of (σ tbl c_11) × ... × (σ tbl c_1i) | ... | C_n of (σ tbl c_n1) × ... × (σ tbl c_nj)]

(* resizable-array reference *)
val mutable f : claz_a resizeArray {idx:"col"} with keybind on cascade ⇝
   argument: ?(f:claz_a resizeArray = new resizeArray [ ])
   field-expr: val mutable f : claz_a resizeArray = f
   action: match (get_metadata keybind) with
         |Key k -> eq (σ tbl col) int; assert_equal k.to_table tbl; exists (π claz_a) tbl k
         |Bind (k_1,k_2) -> eq (σ tbl col) int; κ k_1 k_2 tbl (π claz_a)

(* tuple *)
val mutable f "col_1,...,col_n" : (τ_1 × ... × τ_n) ⇝
   argument:
      ~(f:((σ tbl col_1) × ... × (σ tbl col_n)))
   field-expr: val mutable f : (τ_1 × ... × τ_n) = f
   action: no additional action

(* anonymous object *)
val mutable f : <f_1 "c_1" : τ_1; ...; f_n "c_n" : τ_n> ⇝
   argument:
      ~(f :<f_1:(σ tbl c_1); ...; f_n:(σ tbl c_n)>)
   field-expr: val mutable f:<f_1:τ_1; ...; f_n:τ_n> = f
   action: no additional action

(* associated reference *)
val mutable f : claz_a with keybind on cascade ⇝
   argument: ~(f : claz_a)
   field-expr: val mutable f : claz_a = f
   action: match (get_metadata keybind) with
         |Key k -> assert_equal k.from_table tbl;
             exists tbl (π claz_a) k;
             assert_false
               (is_optional (σ tbl k.from_column))
         |Bind b -> raise Exception

(* optional associated reference *)
val mutable f : claz_a option with keybind on cascade ⇝
   argument: ?(f : claz_a option)
   field-expr: val mutable f : claz_a option = f
   action: match (get_metadata keybind) with
         |Key k -> assert_equal k.from_table tbl;
             exists tbl (π claz_a) k;
             assert_true
               (is_optional (σ tbl k.from_column))
        |Bind b -> raise Exception

(* resizable-set reference *)
val mutable f : claz_a resizeSet with keybind on cascade ⇝
   argument: ?(f:claz_a resizeSet = new resizeSet [ ])
   field-expr: val mutable f : claz_a resizeSet = f
   action: match (get_metadata keybind) with
          |Key k -> assert_equal k.to_table tbl;
                    exists (π claz_a) tbl k
          |Bind (k_1,k_2) -> κ k_1 k_2 tbl (π claz_a)
```

Figure 3.5: ORM syntactic extension translation

```
module Numeric = struct
  type t = NumPS of int * int * Num.t (* numeric value with precision and scale *)
         |Num of Num.t (* without precision and scale *)

  (* plus calculates the addition of two numbers *)
  let plus t1 t2 =
    match (t1, t2) with
    |(Num t1), (Num t2) -> Num (Num.add t1 t2)
    |(Num t1), (NumPS (p, s, t2)) -> NumPS (p, s, (Num.add t1 t2))
    |(NumPS (p, s, t1)), (Num t2) -> NumPS (p, s, (Num.add t1 t2))
    |(NumPS (p1, s1, t1), (NumPS (p2, s2, t2)) ->
        let p, s = (min p1 p2), (max s1 s2) in
        NumPS (p, s, (Num.add t1 t2))

  (* we omit other operations *)

end
```

Database Type	OCaml Type
INTEGER	int
INT4	int32
INT8	int64
FLOAT DOUBLE	float
CHAR	char
VARCHAR TEXT	string
BOOLEAN	bool
MONEY	Numeric.t

Database Type	OCaml Type
Nullable column of type τ	τ option
NUMERIC(precision, scale) NUMERIC(precision) NUMERIC	Numeric.t
DATE	Calendar.date
TIMESTAMP	Calendar.timestamp
TIMESTAMP with TIMEZONE	Calendar.timestamptz
TIME	Calendar.time
TIME with TIMEZONE	Calendar.timetz
TIME INTERVAL	Calendar.time_period

Figure 3.6: Type Mapping between relational database and OCaml language

3.2.3 Compile-time code generation

CamlP4 [dR03] is invoked before the compilation and parses OCaml code into an abstract syntax tree (AST), then allows programs to traverse or transform the AST. Because of its integration with OCaml, the newly generated AST can be fed into the compiler directly without additional processing. Such an approach allows us to extend the language syntax (e.g.embedding the mapping metadata in program code) without kernel compiler modification, and let the compiler guarantee the type safety of generated or transformed code.

Moreover, compile-time code generation enables to provide a customised, explicitly typed, object-relational interface, rather than a generic interface, sacrificing the compile-time type safety. For instance, Hibernate provides save/update/delete/load interface, which takes untyped Objects as arguments or return values, relying on the programmers to perform type casting. In contrast, if explicit functions for various entities were provided, such as $load\_t : session \rightarrow t_{id} \rightarrow t$, which loads entity of type t by passing the identifier that is of type t_{id}. Unsafe type castings would thus be removed from the programming code. Using compile-time code generation makes this achievable.

3.2.4 Additional runtime schema checking

With a loosely coupled database approach, other clients can modify the database schema at any time. If this happens after schema extraction and compilation, but before the compiled program is executed, type incompatibilities might be introduced. In order to minimise the risks of executing programs on an mismatched database schema, the design of our ORM framework involves runtime schema checking, which is invoked the first time a *session* (or *DataContext*) instance is initialised,

with the aim of detecting the mismatch errors as early as possible. This is clearly not a complete solution as the database schema can even be changed by another program during the application's execution.

3.3 Proxy for lazy loading

Navigational queries allows programmers to navigate from one entity to its related entities without requiring application programmers to code explicit loading actions. However, if all relationships were chased and loaded eagerly, rather than lazily, loading an entity could have the effect of loading a large graph of related instances. In the worst case, it might load all of the data from the database.

Proxy based lazy loading provides a solution to this problem. A proxy instance is returned, instead of an initialised object, when a loading action is invoked. Each proxy is essentially a lazy loading cache for a single persistent object, it interrupts and defers the process of object loading until it is necessary.

To achieve lazy loading, we take a similar approach to Hibernate, using proxies for entities and collections. The entity proxy has the same interface as its delegate class, and dispatches actions to its concrete target if the target has been loaded. If the target has not yet been loaded, it carries out the load first and then dispatches the action. The collection proxy, also called a persistent collection, is a subclass of the corresponding standard object-oriented collection, and dispatches actions to its concrete target as per a normal proxy. However, it also takes on the responsibility for loading/removing/persisting collection elements from the underlying database.

3.3.1 Entity proxy

Instead of proxy-based lazy mechanism, lazy evaluation (also known as delayed evaluation) is used, particularly in functional programming languages, to defer the execution of expression until it is requested [Hud89]. That is, in a lazy programming language, a statement like `x=f a` assigns the result of function application to variable x, but what actually is in x is can be a lazy thunk (a memory structure), that is not evaluated until its value is needed in some other evaluation or forced by the user. The first time a lazy thunk is forced, the original expression is evaluated; the evaluated value is returned as the result of forcing the lazy thunk, and cached within the thunk itself. So the second and subsequent times the thunk is forced or requested by other evaluation, the cached value is simply returned.

Programming languages (e.g.Haskell) delay evaluation of expressions by default, and some others provide functions or special syntax to delay evaluation. OCaml belongs to the second category, which supports lazy mechanism by explicitly suspending the computation using the `lazy` keyword. More generally, expressions wrapped by `lazy` define polymorphic value of type `'a Lazy.t`, a deferred computation that has a result of type `'a`. Such lazy expressions could be forced to evaluate by using the function `Lazy.force`.

Lazy evaluation is used to create calculable infinite lists without infinite loops or size matters interfering in computation. However, the style of lazy evaluation in OCaml requires users to explicitly invoke `Lazy.force`, to evaluate the lazy expression and results in an explicit type conversion (i.e. it takes `'a Lazy.t` as parameter and returns `'a`). This, therefore, cannot be used to provide a transparent lazy loading layer. In other words, a lazy expression cannot be directly used in the place where its return value is used.

Proxy-based lazy loading achieve the same effect, but can do so transparently. ORM frameworks, like Hibernate and JDO, adopt the virtual proxy approach, which is to create a proxy delegating a persistent object not yet loaded from a database. Proxies, in general, have the same interface with an ORM object, and holds an initialised or uninitialised target referencing the real object. When any operation on the proxy is invoked, it will initialise its target if not initialised before and then pass the request to the target object (Figure 3.7). In object relational mapping, proxies typically remain uninitialised until they are really needed in the program.

The implementation of proxies in OCaml is necessarily different from that used in Java/C# based frameworks, which typically rely on techniques like *type reflection*, *runtime method interception* and *runtime byte code generation* to generate and dynamically load proxy classes at runtime.

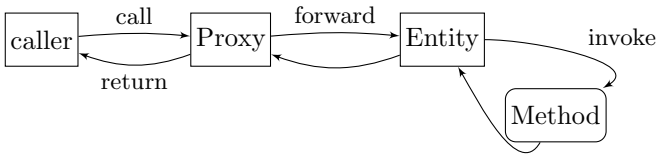


Figure 3.7: Proxy, Entity and Method

Because of the lack of these runtime techniques, proxies in OCaml are implemented by using an alternative approach of statically generating proxy declarations at compile time.

Because each persistent object class, or entity, is different from the others, it is necessary to generate individual proxy classes for each of them. Figure 3.8 gives a sample proxy class for the `order` class, which uses `id` as the identifier. Class `orderProxy` is a higher-order class, which takes `id` and function `loadFun` as arguments, where `loadFun` is the function used to initialise or load order instances from databases. It is worth noting that class `orderProxy` doesn't inherit from `order`, even though `orderProxy` could still be used as a subclass of `order`, based on the fact that OCaml uses structural sub-typing, in which type compatibility and equivalence are determined by the type's structure, and not through explicit declarations. This contrasts with nominal sub-typing (like Java and C#), where comparisons are based on explicit declarations or the names of the types. Structural sub-typing is arguably more flexible than nominal sub-typing, as it permits the creation of ad hoc types and interfaces [Pie02]; in particular, it permits creation of a type which is a super-type of an existing type T, without modifying the definition of T.

```
class order id date details =
object
  (* identifier of the class *)
  val mutable id : int = id
  val mutable date : date = date
  val mutable details :
       detail list = details

  method id = id
  method set_id id' = id <- id'

  method date = date
  method set_date d = date <- date

  (* we omit other methods *)
end

let make_proxy id session =
  let proxy = new orderProxy
      id (loadorder_fromSession session)
  in
  (* cast proxy to order *)
  proxy :> order
```

```
class orderProxy id loadFun =
object
  val mutable id : int = id
  val mutable target : order option = None

  (* private method to obtain target *)
  method private get_target =
     match target with
     |Some t -> t
     |None -> let t = loadFun id in
              target <- Some t;
              t

  method id = id
  (* changing id affects the target *)
  method set_id id' = id <- id'; target <- None

  (* dispatch method invocation to the target *)
  method date = self#get_target#date
  method set_date = self#get_target#set_date

  (* other methods are similar to date and set_date *)
end
```

Figure 3.8: Standard Class and Corresponding Higher-order Entity Proxy

Proxy classes like `orderProxy` contains fields for class identifier and target instance, where target instance is of optional type: `None` means the target object has not been initialised or loaded; `Some x` implies the target instance is loaded and cached as value `x`. `get_target` is an internal, private method used to obtain the delegated object for method invocation dispatch; in particular, it returns the object in the case the target has been loaded, otherwise it loads the object and then returns it. The method for changing the identifier (i.e. `set_id`) clears the cache of the target, as

the value of identifier determines the target object, the modification of which normally implies the change of the delegated object. Function `loadFun` takes the responsibility of loading the object from the database according to the object identifier. Because the same object can be requested, independently, more than once in a session, we must ensure that the same in-memory object is returned in each case. Otherwise, we could end up with multiple clones of the same object in memory and dirty data checking could not reliably reflect changes consistently back to database. Hence object proxies are used with a cache mechanism. Thus instead of loading the object directly from the database, the `loadFun` could search the cache system before each real database hit.

3.3.2 Collection proxy

A collection proxy, also called a persistent collection, is a subclass of a standard object-oriented collection, and takes the responsibility of loading, removing or persisting collection elements from the underlying database. Like entity proxies, a persistent collection maximally defers the initialisation process. For example, in an ordering system, each order consists of a number of order details, which is designed as a collection, named *details*, using either the *List* or *Set* structure. Instead of loading all related details eagerly, the call of `getDetails` on an order instance returns a persistent collection, which would later invoke the loading process on demand. Moreover, even if some actions on the collection are invoked, it can sometimes remain uninitialised (no loading action for the concrete order details). Suppose we iterate through a list of orders in order to obtain the number of details saved in individual order, the invocation of `size` method on the persistent collection could efficiently issue an aggregate SQL query, similar to `SELECT count(*) from orders where id = ?`, rather than loading all concrete data and then performing the in-memory calculation. Such a persistent collection (proxy) significantly reduces the cost of I/O operations between the program and the database, and can be achieved without the user's awareness.

The standard OCaml list and set are not object based, and therefore are not compatible with the proxy mechanism. Hence we need to design a suitable object based persistent collection data type. Our design is for polymorphic, object-oriented, imperative collections. We define the module interface and concrete class type for resizable arrays in Figure 3.9 and 3.10. Similar interfaces could be defined for resizable sets. *ResizeArray* is a *List*-like, but imperative style module, which abstracts the internal type representation (`'a t`) of resizable arrays, and defines a set of imperative (with side-effect) operations on the abstract type, including conversion, manipulation, search, iteration, etc. Moreover, in order to meet the requirements of functional programming, it defines pure functional operations working on type `'a t` without side-effects. Because of the abstraction of type t, such interfaces could be implemented by using various data structures. We define, in Figure 3.10, a polymorphic, object-oriented class type, using the standard list as internal representation, for achieving proxy based lazy loading.

Persistent collections include a target object of option type that can be initialised from the database on demand, and also include an event cache (operation queue) to delay collection initialisation because of method invocation. Specifically, the persistent resizable array, as an example (Figure 3.11), employs the standard list as an internal representation; the target object is set to `None` when the proxy is initialised and there is no elements passed to the proxy; methods encapsulated in the proxy collection can be divided into three categories based on whether the invocation of this method will, may or will not initialise the target collection from the database.

- No initialisation invoked. Because of the operation queue, actions like `add`, `remove` on an uninitialised persistent collection are cached in the queue, which postpones the action until the target collection is initialised or until the session (the ORM container) object flushes modifications back to the database.
- May cause initialisation. Some actions (e.g. `size`, `mem`) may require an initialised target collection, but not in all cases:
    a. The invocation of `size` on an uninitialised collection actually issues an aggregate SQL query to the database, rather than loading data immediately. However, when there are `add/remove` actions postponed in the operation queue, the SQL result returned from

```
module ResizeArray : sig
  type 'a t (** abstract, polymorphic type *)

  val make : 'a list -> 'a t
  (** make a resizeArray from a normal list *)
  val contents : 'a t -> 'a list
  (** return the contents of a resizeArray. *)
  val empty : unit -> 'a t
  (** return a new empty resizeArray *)
  val length : 'a t -> int
  (** return the number of elements *)
  val hd : 'a t -> 'a
  (** return the first element, may raise [Empty_list] *)
  val tl : 'a t -> 'a t
  (** return a new resizeArray without the first element, may raise [Empty_list] *)
  val flatten : 'a t t -> 'a t
  (** concatenate a collection of resizeArrays. *)

  (** {ResizeArray Manipulation} *)

  val add : 'a -> 'a t -> unit
  (** add an element at the end *)
  val push : 'a -> 'a t -> unit
  (** add an element at the head *)
  val pop : 'a t -> 'a
  (** remove and returns the first element, may raise [Empty_list] *)
  val append : desc:'a t -> 'a t -> unit
  (** add all elements of the 2nd resizeArray to the 1st one *)
  val filter : ('a -> bool) -> 'a t -> unit
  (** remove all elements that do not match the specified predicate *)
  val remove : 'a -> 'a t -> unit
  (** remove an element from the resizeArray, may raise [Not_found] *)

  (** {Iterators} *)

  val iter : ('a -> unit) -> 'a t -> unit
  (** apply the given function to all elements of the resizeArray *)
  val map : ('a -> 'b) -> 'a t -> 'b t
  (** construct a new resizeArray by applying the function to all elements *)
  val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
  (** apply a function to the elements of resizeArray in a fold_left style. *)
  val fold_right : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  (** apply a function to the elements of resizeArray in a fold_right style. *)

  (* we omit functions for association, searching, scanning, sorting,
     and similar functions working on two ResizeArrays *)

  (** {Functional Manipulation} *)
  (** For all functions ended with [_F], there is no side-effect when being invoked. *)

  val add_F : 'a -> 'a t -> 'a t
  (** return a new resizeArray by adding an element at the end *)
  val push_F : 'a -> 'a t -> 'a t
  (** return a new resizeArray by adding an element at the head *)
  val append_F : 'a t -> 'a t -> 'a t
  (** concatenate two resizeArrays *)
  val remove_F : 'a -> 'a t -> 'a t
  (** construct a new resizeArray by removing an element, may raise [Not_found] *)

  (* we omit many more functions *)
end
```

Figure 3.9: Module Interface for Resizeable Array

```
class type ['a] resizearray =
  object ('b)
    method elements : 'a list
    method set : 'a list -> unit
    method contents : 'a list

    method add : 'a -> unit
    method push : 'a -> unit
    method pop : 'a
    method remove : 'a -> unit
    method remove_all : 'a -> unit
    method remove_if : ('a -> bool) -> unit
    method filter : ('a -> bool) -> unit
    method transform : ('a -> 'a) -> unit
    method rev : unit
    method clear : unit

    method is_empty : bool
    method length : int

    method first : 'a
    method last : 'a
    method nth : int -> 'a
    method hd : 'a
    method tl : 'b

    method exists : ('a -> bool) -> bool
    method for_all : ('a -> bool) -> bool
    method find : ('a -> bool) -> 'a
    method find_all : ('a -> bool) -> 'b
    method mem : 'a -> bool
    method memq : 'a -> bool
    method partition : ('a -> bool) -> 'b * 'b

    method fold_left : 'c.('c -> 'a -> 'c) -> 'c -> 'c
    method fold_right : 'd.('a -> 'd -> 'd) -> 'd -> 'd
    method iter : ('a -> unit) -> unit

    method sort : ('a -> 'a -> int) -> unit
    method fast_sort : ('a -> 'a -> int) -> unit
    method stable_sort : ('a -> 'a -> int) -> unit
end
```

Figure 3.10: Polymorphic, Object-oriented ResizeArray Type

the database server cannot reflect the accurate size, which requires us to initialise the collection to calculate the true size.

b. In contrast, `mem`, checking the existence of certain element, normally requires the initialisation of collection. However, in the case when the element has already been involved in the operation queue, either in an `add` or `remove` action, the initialisation is redundant and can be avoided.

- Invoke initialisation. Actions, such as `iter`, `map`, etc, require initialising the collection from the database before execution.

Generics is a programming language design element for providing type safe containers parameterised by the type of the contained elements. Many programming languages support some variety of generics features, such as the C++ templates, Java and C# generics. In OCaml, functors provide many of the solutions [FL04, CFS08, BSSS06] that people use generics for, while not involving additional problems (e.g. type erasure in Java generics [BOSW98], which imposes some restrictions on, among other features, object and array creation). In the case when complex collections and operations are difficult to handle via polymorphism alone, functors become the natural expression for the problem.

Functors are higher-order, parametrised modules, which take modules or other functors as parameters. They provide the capability of creating generic modules; the application of a functor generates a concrete module at compile time. If we have a module that implements some operations on a collection of objects of one data type, we don't want to write new modules when an additional data type is introduced. We get avoid the extra work by using functors to create a generic collection module that takes, as argument, a module defining the data type and ordering function of the collection elements.

The persistent collection in our case (Figure 3.11) is designed as a functor with an argument of signature `LOAD`, which declares the type of element, collection key, session, and functions for comparison (`equals`), retrieving data from database (`load`), and calculating the size of collection elements (`size`). The result of the functor application is a module providing a type-specific persistent collection class with various operations.

Suppose we define a module `OrderLoad` satisfying the signature `LOAD`. The following statement generates a concrete persistent array module, named `OrderPersiArray`, for the order class, where type `OrderLoad.key` represents the key of the persistent collection, itself consisting of three items: the name of corresponding class and field identifying where this collection is referred to, and the identifier value of the instance that owns this collection. `OrderLoad.sess` is the type of the ORM session instance, i.e. the database connection.

```
module OrderLoad : LOAD = struct
  type t = order
  type key = {classname: string; field: string; id: int}
  type sess = Session.sess

  (* implementation of functions: equals, load, size *)
  ...
end

module OrderPersiArray = PersistentArray (OrderLoad)
```

3.3.3 Identifying proxies

When an entity is requested by the application program, a proxy will normally be returned in order to support lazy loading. Hence we want the programmer to be unaware that the object he is manipulating is actually a proxy to a true entity rather than a true entity. This is done by having the proxy inherit from the class of the entity that it refers to. Hence a proxy object *is-a* target class object. At various points, however, we need to take different actions in a session depending on whether the object we are acting upon is a proxy or a true entity.

```
module type LOAD = sig
  type t (** element type *)
  type key (** collection key *)
  type sess (** session type *)

  val equals : t -> t -> bool
  val load : sess -> key -> t list
  val size : sess -> key -> int
end

module PersistentArray (L : LOAD) = struct
  type 'a action = (** action type *)
       [`APPEND of 'a
       |`REMOVE of 'a
       |`REMOVE_ALL of 'a]

  (** object-oriented persisent array class *)
  class ['a] t sess role key ?elements () =
  object (self : 'b) constraint 'a = L.t
    val mutable sess : L.sess = sess
    val mutable key : L.key = key
    val mutable operation_queue : action list = []
    val mutable dirty = false
    val mutable target : L.t list option = elements

    method is_initialized = match target with
         |None -> false
         |Some _ -> true

    method private get_target = match target with
         |Some c -> c
         |None -> raise Uninitialized

    method private set_target t' = target <- Some t'

    (** operation queue *)

    method private queue_operation act =
        operation_queue <- (operation_queue @ [act]);
        dirty <- true

    method private is_operation_queue_enabled =
      (not (self#is_initialized)) &&
         (self#is_sess_open)

    method private has_queued_operations =
      not (List.is_empty operation_queue)

    method private perform_queued_operations =
      let t' = self#get_target in
      let perform = function
        | `APPEND x ->
           self#set_target (List.append t' [x])
        | `REMOVE x ->
           self#set_target (List.remove x t')
        | `REMOVE_ALL x ->
           self#set_target (List.remove_all x t')
      in
      List.iter (fun x -> perform x) operation_queue;
      operation_queue <- []
  end
end
```

```
(** class t cont. *)

(** initialize collection *)

method private initialize =
  if (not (self#is_initialized)) then(
      assert_conn_open sess;
      self#set_target (L.load sess key);
      self#after_initialize)

method private after_initialize =
  initialized <- true;
  self#perform_queued_operations

method private read = self#initialize
method private write =
    self#initialize; dirty <- true

(** operations on persistent array *)

method add = fun e ->
  if self#is_operation_queue_enabled then(
     self#queue_operation (`APPEND e)
  )else if (self#is_initialized ||
            self#is_sess_open) then(
     self#write;
     self#set_target (self#get_target @ [e])
  )else raise SessionClosed

method size =
  if (self#has_queued_operations ||
       self#is_initialized) then(
     self#read;
     List.length self#get_target
  )else
     L.size sess key

method mem = fun e ->
  try
    List.iter (fun act ->
      match act with
      |`APPEND x ->
        if (L.equals x e) then raise Found
      |`REMOVE x ->
        if (L.equals x e) then raise Removed
      |`REMOVE_ALL x ->
        if (L.equals x e) then raise Removed
    ) operation_queue;
    raise Not_found
  with Found -> true
      |Removed -> false
      |Not_found -> self#read;
        List.exists (fun x -> L.equals x e)
           self#get_target

method iter = fun f -> self#read;
  List.iter f self#get_target

(* we omit other methods *)
```

Figure 3.11: Polymorphic, Higher-order Module for Collection Proxy

For example, consider the `save` action. This provides a unified interface (despite different implementations) for persisting entities or proxy instances into the underlying database. In the following code, lines 1–2 load a student (the orm class is defined in Figure 3.3) by identifier (in this case the "100") and modifies her email. Because of lazy loading, `stu01` is returned as a proxy instance on line 1 but this proxy is actually instantiated (the corresponding true entity is loaded and the proxy made to refer to it) by line 2. Line 4 creates a new student entity, and this is a true entity, rather than a proxy. In the last two lines, we persist these two student instances. In fact, the programmer need not include line 6, as, at the end of the session, the Qanat system will detect that the persistent entity referred to by the `stu01` proxy has been modified and will itself invoke `Session.save` on it if the programmer has not already done so. Nevertheless, this invocation is legal and, whether invoked by the programmer or the system, will be executed. For the proxy instance, `stu01`, aside from persisting the proxy target and updating its cache, a check has to be made to see if the proxy is already cached and associated with the current session. In this case, since the proxy was loaded in the current session, it will be found to be so associated. However, if the proxy had been a detached proxy loaded in a different previous session, additional work must be carried out to associate and cache the proxy into the current session. The purpose of this work is to guarantee that the same proxy instance is returned the next time the same student record is requested. In contrast, for the pure entity case, the entity must be cached and saved to the database, but no proxy cache is affected.

```
1 let stu01 = Session.load{student} session 100 in
2 stu01#set_email "alice@cs.bham.ac.uk";
3
4 let stu02 = new student ~name:("tony", "wilson") ~email:"tony@cs.bham.ac.uk" () in
5
6 Session.save{student} stu01;
7 Session.save{student} stu02
```

Thus we need a reliable method for the system to be able to distinguish between proxies and true entities. In Java, we could use runtime type reflection to provide the answer, but this option is not open to us in OCaml. However, it is necessary to distinguish this difference when objects are persisted to the database, which only works on initialised entity instances instead of on proxy instances. The lack of *type reflection* in OCaml makes it difficult to distinguish proxy instances and entity instances at runtime, since type information is no longer present in the compiled code after compilation in OCaml. Thus building an object-relational mapping in OCaml requires the framework to monitor the status (i.e. type) of proxy instances.

While we could modify the entity class to add a method, overridden in the associated proxy class, that would identify the status of the object, this would break our principle of providing transparent persistence. The programmer would have to be aware that we were adding this extra method, so that he or she does not accidently or on purpose interfere with it. Instead we use a *hash set* data structure, to which we add references to all initialised proxies. Checking the status of an entity reference, therefore, is simply a question of looking it up in this hash set. One problem remains: this hash set cannot be stored in the session, but must be global and maintained for the lifetime of application.

When a database transaction finishes, the related session is closed. However, entity references can remain. Such entity references are called *detached*, as they are not connected to any session and cannot, without such a connection, be saved to the database if they are updated. Nonetheless, detached objects are important and feature heavily in multi-layer applications: the control and model layers obtain an object from the database within a transaction, the transaction finishes, the object, now detached, is passed to the view layer, which presents it to the end user. Further, the end user may then cause the detached object to be modified, whereupon it is passed back to the control and model layers, which then start a new transaction, re-attach the object to the session and save the change to the database. Thus, since a proxy initialised in one database transaction could still be passed to and used in the others, we have to be able to recognise this detached but initialised proxy as such. Hence the hash set needs to be maintained not just for the lifetime of one session, but for the lifetime of any proxy, which could be the lifetime of the application. Hence we must make the hash set a global data structure in the application.

However, since this hash set is now long lived, we need to concern ourselves with the problem of memory. Even though OCaml supports garbage collection, and this garbage collector is automatically invoked to release memory allocated to unreachable objects, objects saved in the global hash set, even if there is no other reference to them in the program, cannot be reclaimed because of the permanent reference from the hash set.

In order to avoid memory leaks, we use weak pointers [Got74, Haiml, CD08] as an alternative, in the form of a weak hash tables, in which both keys and associated values are weakly stored. This gives us a smart cache, in which the values that are referenced by weak pointers can be reclaimed if they are no longer referenced by other entities through non-weak references.

OCaml language has had a built-in implementation of weak pointers since 1997 [Ler97]. The 3.10.2 version of the OCaml system [LDG+07] has achieved significant improvement in this aspect of weak pointer implementation. For instance, Hashconsing [FC06] in OCaml, based on the provided weak arrays and weak hash tables, now works reliably and is employed in heavy duty applications [CC06, BDD07, dt08]. The OCaml weak hash table library allows arbitrary functions to be used as the equality and the equality-compatible hash function on keys. In practice, it is presented as a functor to be applied to a module that provides these functions. To meet our requirements of the smart cache for various proxies, we define *P*, a module of objects with a `Hashtbl.HashedType` interface (i.e. providing a type `t` and functions `hash` and `equal`). The weak hash table of proxies can be created with:

```
module P = struct
  type t = < >
  let hash = Hashtbl.hash
  let equal = (==)
end

module ProxiesWeakHashtbl = Weak.Make (P)
```

where `t` is the type of objects without explicit methods, analogue to Object in Java. `hash` and `equal` are ordinary hash functions and physical comparison functions. Performance testing has shown that garbage collection works on the weak hash table, and the use of memory is kept to a reasonable size.

3.4 Session objects

ORM frameworks typically provide the mapping mechanism between relational databases and programming instances through an entity container, called a *session* (or *DataContext* in LINQ). A session is a wrapper for a database connection and a container of all persistent object instances used in a use case execution. It works like a caching system but provides more functionality; a session not only manages the status of persistent objects, but also maintains data consistency between the database and the programming environment, and persists modifications back into the database automatically when the session is closed.

3.4.1 Session structure

The session, firstly, acts as a caching mechanism. When an entity is requested by the program, the session first performs a lookup in the cache. If the specified entity with valid state exists, this instance is returned. Otherwise, an entity proxy will be returned, which defers the database hit until an operation invocation happens on the proxy.

To support this behaviour, the session object is designed to store a series of key-value data structures (e.g. hash table or hash map) as the cache for persistent entities, proxies delegating to persistent entities, and persistent collections owned by entities, respectively. Each persistent entity and proxy is identified by an entity key, which consists of class information and primary key value (i.e. identifier of the entity); persistent collections are identified by collection keys, which record the role, describing the relationship between the owner and this collection (i.e. the owner class and the corresponding class field), and the specified foreign key information. Critically, each

persistent entity and collection is associated with an entry instance, which is an data snapshot of the entity/collection as it currently appears in the database. This entry instance can be compared with the current version of the object instance in use by the program to determine if the object is dirty, i.e. differs from the database copy and therefore needs to be written back to the database.

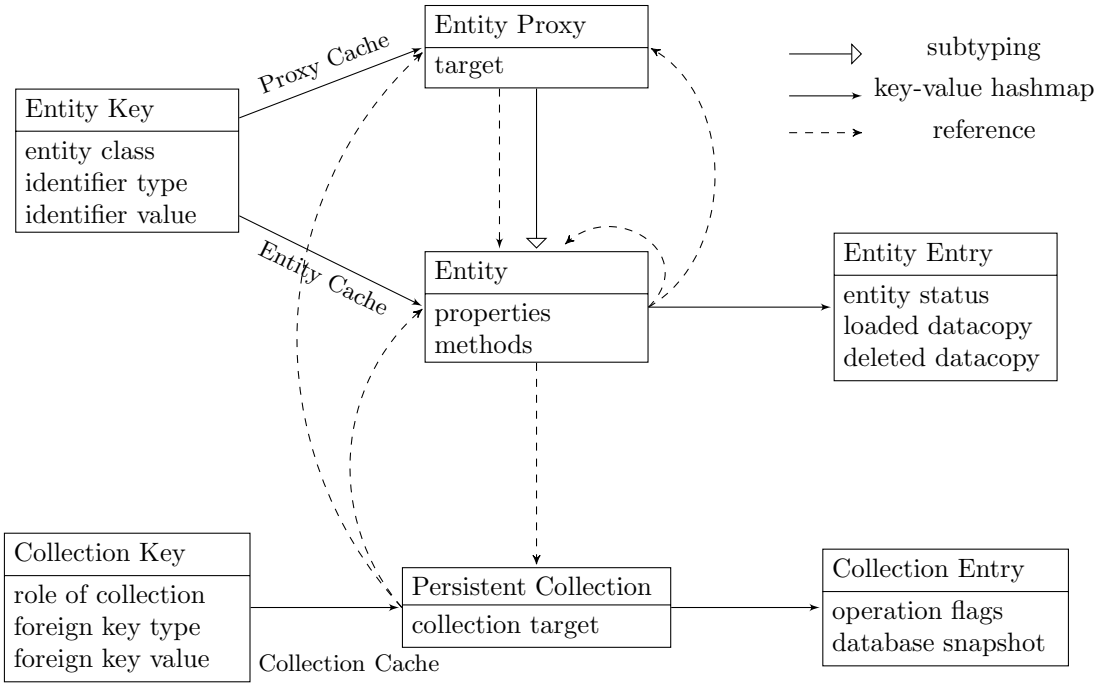


Figure 3.12: ORM Session Cache Structure

Figure 3.12 depicts the structure of the session cache. In practice, entity and entity proxy may have the same entity key, in the case that both of them represent the same data from the database, i.e. the proxy instance delegates to this entity. Entity, entity proxy and persistent collections can have references to each other based on the business logic requirements. The session maintains an entry instance for each entity and persistent collection, which determines the operations that should be performed during a session flush. The entity entry includes the status and two data copies of the entity; the status is the value of:

Managed: indicates that the current entity is managed by the session, and is a persistent object, modifications on which would be written back into the database.

Deleted: implies this entity has been deleted, and should be removed from the database when flushing the session.

Loading: is set before the loading action of an entity occurs, and is used as a flag to avoid recursive loading. It is modified to **Managed** when the loading finishes.

Saving: similar to **Loading**, but is used to avoid recursive saving actions happening on the same instance, in the case that managed entities construct a cyclic relationship via references.

The loaded data copy remains consistent with the data in the database, and is used to perform dirty data checking. When a newly created in-memory entity instance is saved into the session, but has not yet been flushed to the database, the loaded data copy is marked as empty.

The deleted data copy is the exact contents of the entity when it was removed from the session.

In the case of deleting a cyclic relationship from the database, the deleted data copy is used to keep the status after breaking the relation chain (we discuss this in more detail in section 3.4.4).

Aside from the database snapshot that records the original loaded elements in the collection, the collection entry includes an operation flag, which is set during the flush process, indicating

whether this collection proxy is going to be updated, in the case that elements are added into or removed from the collection, removed in the case that this collection is no longer referenced by any persistent entity, or recreated, when the owner of the collection is modified (e.g. move the collection from one entity to another one).

The key-value cache is implemented as a hash table. Unlike the hash table in Java, in which every object class is a sub-type of the base Object class and, so can be inserted into a hash table and extracted with down casting based on knowledge of original type obtained through reflection, OCaml objects do not have a common base type and this limits the hash table to be explicitly typed. Therefore, we unify various types of values as a union, or variant, type that can be stored into the hash table. We can then use pattern matching to extract the real object, rather than rely on reflection and down casting. A simple cache, for example, can be defined as:

```
type key = Key1 of int | ...| KeyN of string
type value = Entity1 of classA | ... | EntityN of classN
let hashtbl : (key, value) Hashtbl.t = Hashtbl.create size
```

Compared to the entity cache, keys of persistent collections include not only the identifier of the collection owner, but also the role of the collection, in order to distinguish multiple collections existing in the same entity. The role of a collection instance identifies where the collection is located, including the name of class and field, as demonstrated in the following code:

```
type role = {classname: string; filedname: string}
type uniontype = Int of int | ... | String of string
type collectkey = {role: role; key: uniontype}
type collectvalue = CollectionA of classA collection | ...
                  |CollectionN of classN collection
```

3.4.2 Life cycle of entity

According to the state of data in relational databases, entities can be categorised as transient objects, which exist in memory but not in the database, persistent objects managed by session objects, and detached objects, which have a copy in the database but are not, or no longer, managed by the current session object.

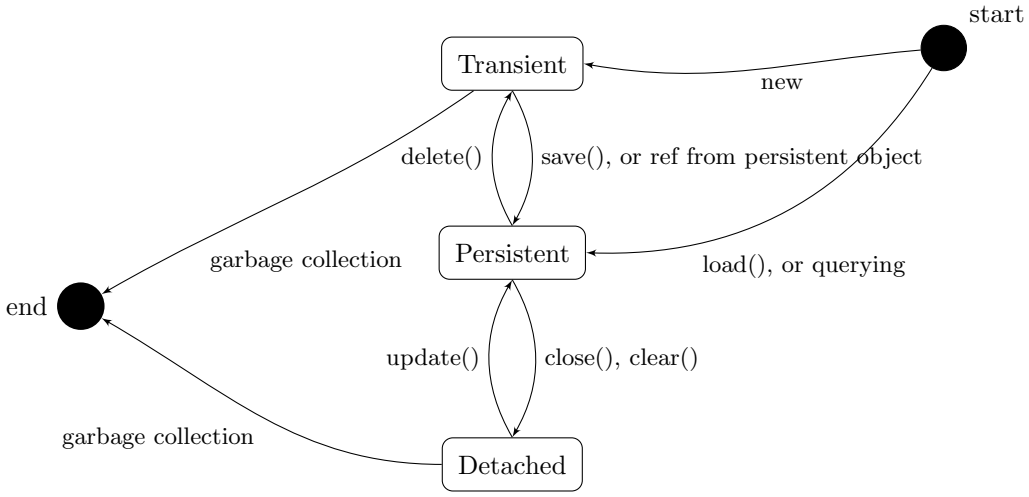


Figure 3.13: ORM Entity Lifecycle [BK04]

The session module provides support for transformation between different states via a load-/save/update/delete interface. Moreover, unlike traditional database operations, these actions on the session objects are normally aggregated, delayed until the session object is about to be closed, and performed in a cascade style (section 3.4.4). Figure 3.13 depicts the transformation between

various states. For example, transient objects, created by the program, could be persisted into the database by invoking the method `save`, or by having references to them added to other persistent objects. Detached objects can become persistent in (managed by) the current session through the `update` interface. In reverse, persistent objects become transient when they are deleted from the session (i.e. the database), or become detached when the session is closed or when the object is evicted from the session. All these objects are finally reclaimed by the garbage collector when they are no longer referenced in the program.

3.4.3 Modification tracking

Qanat, like Hibernate, identifies dirty data and persists these modifications into the database when the session is about to close. Such a mechanism, in contrast with the immediate modification update, reduces the number of, and defers the time of, database hits to session flushing. There are two main approaches to accomplish dirty data checking:

Proxy based method interception: Rather than plain objects, data instances are wrapped as a proxy, thus method invocation on data objects are intercepted; any setter operation on data attributes will trigger the 'dirty' flag, indicating that the data should be written back to the database.

Caching data copy for comparison: This approach keeps a snapshot of the object as it was when loaded from the database. During flushing, the persistence manager (i.e. the session) compares the current object with its loaded snapshot. If differences exist, the specified object needs to be updated in database.

Frameworks like Hibernate adopt the latter technique, maintaining the loaded data snapshot; LINQ provides support for both approaches.

Both approaches have advantages and disadvantages. The interception approach adds a small overhead on every update, but no extra work is required to identify dirty data when the session is flushed. The copy method avoids the update overhead, but at the cost of more work when flushing the session. As far as space costs are concerned, the copy method requires extra memory to store the copies. The interception does require some extra memory to record which objects are dirty, but as both methods need to use proxies for reasons of lazy loading, the extra memory needed by the interception method is small compared to that of the copy method.

To achieve a simpler solution and better performance, we take the same approach as Hibernate for the dirty checking. In practice, as depicted in Figure 3.12, entity entry caches two copies of the data, the loaded data copy and and the deleted data copy. The loaded data copy is the original state, consistent with the record in the database. The deleted data copy is set only when the entity is going to be removed from the session; it is obtained by removing the cyclic relationship from the current data value in order to support cascade deletion (section 3.4.4). During the flushing phase, the session flushing code iterates over each entity in the session, performs dirty data checking by comparing the current data value (or the deleted data copy in the case the object is marked as Deleted) to the cached original value, then determines which database operation should be executed.

3.4.4 Cascade actions

Aside from its role in performance improvement, the session takes responsibility for the following ORM functions:

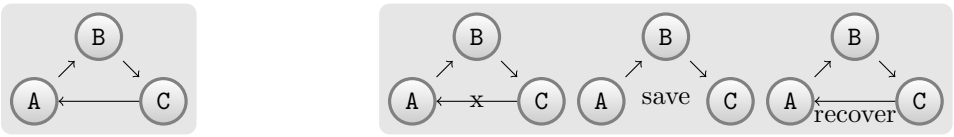
- constructing and initialising entities based on the mapping metadata
- performing load/save/update/delete actions
- maintaining the relationship between entities
- tracking the status of managed entities

Object relational mapping supports operation cascading. This means that programmers can specify, on a relationship by relationship basis, whether operations such as save, delete and update, when applied to the object on one side of the relationship, should also be applied to those on the other. Since associations between entities are defined by foreign key constraints, cascading operations need to obey these constraints. Hence we need to manage the enforcement of referential integrity in the programming language.

Non-cyclic entity reference graphs To avoid breaking referential integrity, database operations must be performed in a specified order. Thus the required operations, when the session flushes changes to the database, need to be sorted and executed in a specified order compatible with the database's referential integrity constraints, i.e insert operations first, then update operations, finally delete operations.

For a specified action, the operation is also cascaded in a certain order. For example, an instance, A will be saved into the database only after all of its referenced instances have been saved. Then the save operation will be cascaded to the elements of its collection references. This order is necessary because elements of collections refer to the collection owner (A) itself. In the delete action, operations are performed in the opposite order.

Cyclic entity reference graphs The situations is more complex when performing cascade actions on entities which have cycles in their reference graphs:



In such situations, operations will be divided into several steps; first break the cyclic relation, and then cascade the save or delete operation on each instance, finally recover the cycle in the case of the save action. For the delete case, there is no need for the last step of relationship recovery, since all the involved instances have been removed from the database.

Before discussing the detail of the cascading actions, we need to know how to check the state (i.e. `Transient`, `Persistent`, or `Detached`) of the object on which the action occurs. The difference between the persistent and transient states is whether or not the object has a corresponding record in the database. In the case that the object has an auto-generated identifier, the session will generate and assign the value of the identifier to the object when saving a transient object. Thus a null-value identifier indicates that the object is transient.

However, if an object has an assigned value identifier, we cannot say the object is persistent or detached, because it may have `assign` or `refer_to` type keys. In such cases, we must search for the object in the session cache. If it is found there, then the object is persistent unless it has been marked as deleted, in which case it used to be persistent but is now transient. If not found in the session cache, the object might be transient or detached, and we must query the database to find out which. Thus the algorithm is as follows:

1. If the object has a null-value identifier, then the object is `transient`.
2. If the object has a non-null identifier, look it up in the session cache:
   (a) Entry found with status 'Managed': the object is `persistent`.
   (b) Entry found and marked as 'Deleted': the object is `transient`.
   (c) Entry found with status 'Saving': the object is `persistent`, even though the save operation is still being processed.
   (d) Entry not found in the session: the object is transient or detached. Execute an SQL query to check if the object exists in the database:
      i. If the object exists in the database, then it is `detached`.
      ii. Otherwise, it is `transient`.

The most common operation in the object-relational mapping is the load function, which queries the database and constructs the entity instance from the query result according to the given identifier value. A load operation could be cascading in the context of providing a solution to the $N + 1$ fetch problem. The $N + 1$ fetch refers to the fact that a common pattern of loading one entity in a program followed loading N related entities can, if precautions are not taken, lead to $N + 1$ separate queries being executed rather than one join query. In order to reduce the number of database hits, we can specify that the load operation could be cascading: i.e. that the loading of one entity should also loads its related entities. This is done by setting the `eager-fetch` property. Setting this property instructs Qanat to generate a join SQL statement for loading the master entity that loads the related entities simultaneously.

Cascading Save

The save operation takes an entity (proxy or entity instance) as argument, persists the entity itself and all its related entities that are specified with the `save` cascade property into the database. Because of the use of proxy based lazy loading, the action of saving involves the process of identifying the instance: whether it is proxy instance (initialised or uninitialised) or entity instance (transient, persistent, or detached). For proxy instances, it is necessary to associate the proxy to the current session, in order to maintain the relationship between proxy and target. For entities with auto-generated identifiers, the value of the identifier is generated and assigned during this process. We give the algorithm for the cascading save in Figure 3.14. In the case the parameter instance is a proxy, it is first re-associated to the current session (i.e. change the session encapsulated in the proxy to the current one), then the target instance, if the proxy is initialised, is treated as a standard entity instance. For uninitialised proxies, the save action terminates after the re-association.

The save action on a transient entity is performed as follows, the session instance generates the object identifier (if necessary) and prepares entity entry, then cascades the save action to entities referred to by the current object before registering this object's insert event in the session's operation list, finally the save action is cascaded to its associated collections. In order to avoid infinite loops when saving a cyclic reference graph, the session sets a `Saving` flag in the entity entry indicating that the save action has started on this entity, and changes the flag to `Managed` when the save finishes. In analogy to the three steps adopted in SQL when saving a cyclic record graph, the session prepares a loaded data copy in the entity entry for this transient object. This loaded data copy acts as an intermediate state that is obtained by nullifying the object's option-type associated single fields (equivalent to breaking the appropriate nullable foreign key constraints in the cyclic graph), and is used to add an insert event in the session's operation list. That is, the object and associated objects are saved into the database without setting nullable foreign keys.

During flushing, the session checks for differences between the entity and its loaded data copy, and, if they differ, an update event is invoked. At this time, if any associated field is nullified in the loaded data copy during the cascade saving, the relation is recovered by the update event. Such a process corresponds to the step of breaking cyclic relations, saving the records, and then recovering the relations. All database operations, including the save, update and delete, are postponed until the session is about to close. To remain consistent with the database, the session performs a post-action: updating the loaded data copy to the current object content for managed objects on which the update action is invoked.

Cascading Update

The update operation reflects the process of persisting data modifications into the database (Figure 3.15). It is similar to the save operation, except that there is no update event generated during this phase, since modifications on persistent entity instances are automatically written back into the database during the flushing phase. The update operation is cascaded to all associated entity instances that have their cascade property set, thus transient objects are persisted into, and detached objects are re-associated, to the current session.

1. If the object is an uninitialised proxy (i.e. the target object is None), re-associate it to the current session by changing this proxy's encapsulated session to, and caching the proxy itself in, the current session, then the function terminates.
2. If the object is an initialised proxy (i.e. the target object is not None), re-associate the proxy to the current session, then perform save action on the proxy target (step 3).
3. If the object is an entity instance, the save is performed according to its state:
   - 3.1. If the object is `persistent`, no action is needed, the function terminates.
   - 3.2. If the object is `transient` or `detached`, start save action on this object, go to step 4.
   - 3.3. If the object is marked as `Deleted`, invoke the flushing process to remove all `Deleted`-marked objects from the database, then start the save action on this object, go to step 4.
   - 3.4. If the object is marked as `Saving` that indicates the save action is invoked cascade in a nested style, the current round of save action terminates.
4. Actions for saving entity instance:
   - 4.1. Prepare entity saving,
     - i. Generate identifier for instance in the case the object has an `auto` key-generator.
     - ii. Initialise an empty entity-entry, the state of which is set as `Saving`, indicating the save action starts.
     - iii. Cache the entity and entity-entry in the session. The entity-entry acts as a placeholder to stop invoking the save action on the same object more than once when cascading saving in a cyclic graph.
   - 4.2. Cascade save action to associated single references on which the cascade property (`all` or `save_update`) is set.
   - 4.3. Register the insert event
     - i. Substitute the instance's associated collections with persistent collection proxies.
     - ii. Create a copy of the object, representing the intermediate state i.e. the object after breaking nullable foreign keys. Such a copy is obtained by setting all option-type single references as None and copying the other fields from the object.

       The purpose of the action, nullifying single references in the copy, is to break the potential cyclic reference chains that break the database referential integrity when trying to save a cyclic graph into the database.
     - iii. Add new insert event for the constructed intermediate-state data copy in the session's operation list.
     - iv. Update the entity-entry by modifying its status to `Managed` and saving the intermediate-state data copy as the loaded data copy in the entry.

       Note, that because option-type single references in the copy is nullified, while the ones in the real entity are not, an update action will be invoked during the flushing time to recover these broken relations. After that, a new data copy that is consistent with the object content is saved as the loaded data copy in the entity-entry. See flushing algorithm in Figure 3.17.
   - 4.4. Cascade save action to elements of associated collections on which the cascade property (`all` or `save_update`) is set.

Figure 3.14: Cascading Save Algorithm

1. If the object is an uninitialised proxy (i.e. the target object is None), re-associate it to the current session by changing this proxy's encapsulated session to, and caching the proxy itself in, the current session, then the function terminates.
2. If the object is an initialised proxy (i.e. the target object is not None), re-associate the proxy to the current session, then perform update action on the proxy target (step 3).
3. If the object is an entity instance, the update is performed according to its state:
   - 3.1. If the object is `persistent`, no action is needed, the function terminates.
   - 3.2. If the object is `transient`, invoke save action on this object, see algorithm in Figure 3.14.
   - 3.3. If the object is `detached`, re-attach and update this object (step 4)
   - 3.4. If the object is marked as `Deleted`, invoke the flushing process to remove all `Deleted`-marked objects from the database, then save this object, see algorithm in Figure 3.14.
   - 3.5. Throw an exception if the object is marked as `Saving`
4. Re-attach and update the detached instance
   - 4.1. Check uniqueness of the instance in the session cache. There should be no cached object that has the same entity key with this instance. Otherwise, throw an exception.
   - 4.2. Substitute all associated collections of this instance with persistent collections.
   - 4.3. Attach the detached instance to the current session: cache entity, and add entity-entry, inside which the status is `Managed` and the loaded data copy remains empty, since at this point, the session doesn't know what is the original data loaded from the database.

     Note, that the real update event is deferred to the flushing time. Because the loaded data copy for detached instances is empty, while the content of the instance is not, it means that the dirty checking always returns true and the content of the instance will be updated into the database no matter whether the instance is really modified or not.
   - 4.4. Invoke the update action on associated references (including single references and associated collection elements) that have their cascade property set to `all`, or `save_update`.

Figure 3.15: Cascading Update Algorithm

Cascading Delete

There are some subtleties specific to cascading the delete operation (Figure 3.16). In spite of the fact that the objects are going to be deleted, such cases require loading associated entities and initialising proxies. In order to maintain the consistency between session and relational databases and avoid unexpected deletion in databases, entities need to be loaded into the session cache before cascading the delete action to the associated entities or collection elements. Rather than being deleted immediately from the database, these entities are marked as deleted in the session cache, which defers the delete action until the flushing time. Each deleted entity is associated with a deleted data copy in its entity entry, which represents the data value in the database when the record is about to be deleted. The deleted data copy can be constructed by copying content from the instance but leaving option-type single associated references as empty. The purpose of this nullifying option-type associated references is to break the potential cyclic relations that cause the delete action fail when trying to remove cyclic records from the database directly. During the flushing process, the session invokes an update event before the delete action to synchronise the table record to the values of the deleted data copy; this removes nullable foreign keys in the record, preparing for the delete action.

Object-relational mapping frameworks provide the facility of dirty tracking, and automatically persist data modifications into the database. Moreover, in order to reduce the database hits, operations, including save, update and delete, are normally cached in an operation queue and postponed

until the synchronisation is necessary. Such a synchronisation between the ORM session and the underlying database (we also straightforwardly call it *flushing*, since only database writing happens during this process) is typically performed when the session is about to close, or occasionally when trying to save a deleted object into the session. A flushing process, in essence, starts by performing the dirty checking on each managed (persistent) entity including those marked as `Deleted`. Then it discovers, and adds into the session, all persistable, transient entities that are referred to by persistent entities, and prepares operations for maintaining relationships. After that, it schedules and executes these queued database operations in a specified order, and finishes the flushing by updating all the involved entity and collection entries. Maintaining relationships includes actions for building new database foreign key constraints, and updating or removing existing ones. To avoid referential integrity problems, the database operations are performed in the order: insert data first, then update data, maintain relationship (remove, update and build), and finally delete data. For concreteness, we give the detail of the flushing process in Figure 3.17.

3.5 Summary

We propose a Hibernate-like, but compile-time type safe, object-relational mapping framework in the functional language, OCaml.

Similar designs to Hibernate are used in order to achieve core functionality. We adopt the data mapper design pattern (section 3.1), which maps object references in in-memory objects to foreign keys in the database, providing a object-oriented navigational querying framework. In section 3.3, we use proxies to support lazy loading of objects from the database on demand, and overload collection classes to manage lazy loading of 1-to-N relationships. Different from Hibernate, which relies on runtime byte code compilation, proxies in our framework are generated through code pre-processing. While the overall design of this part of Qanat is very similar to Hibernate, many of the details are different because of the need to satisfy compile-time type safety and because of the different problems, constraints and opportunities that the use of OCaml, rather than Java, implies. In section 3.4, we describe the design of our session object, which, as the kernel of the ORM framework, takes the responsibility for managing (save, update, delete and load) persistent objects from the database, and reflecting data modifications back into the database automatically.

In order to guarantee compile-time type safety, we introduce a number of approaches in section 3.2. Instead of using an external, literal XML mapping file, we embed the mapping meta in the code by extending the language syntax via CamlP4 pre-processing, and introduce database schema information during compilation, using which the extend ORM class definition is translated into standard language code with proper type information. Moreover, additional runtime schema checking is employed to minimise the risks of executing programs on an a database whose schema has been modified since the program was compiled.

Compared to programming languages like Java, OCaml has the features of imperative, functional, and object-oriented programming paradigms, and our design benefits from functional features, such as type inference, higher-order functions, and parametrised modules (functors). In section 3.3.2, we demonstrated the design of a polymorphic, higher-order persistent collection module.

1. Get the entity instance
   - 1.1. If the entity is a proxy, re-associate the proxy to the current session, then return the target object. In the case the proxy is uninitialised, initialise the proxy then return its target.
   - 1.2. If the entity is a standard object, return it directly.
2. Identify the state of the entity instance.
   - 2.1. If the instance is already marked as `deleted`, the function terminates.
   - 2.2. If the instance is `transient`, throw an exception (transient object cannot be deleted from the database).
   - 2.3. If the instance is `persistent`, perform the delete action (step 4).
   - 2.4. If the instance is `detached`, re-attach it to the current session (step 3).
   - 2.5. Throw exception if the instance is marked as `saving` (unexpected case).
3. Re-attach the detached instance to the current session
   - 3.1. Check uniqueness of the instance in the session cache. There should be no cached object that has the same entity key with this instance. Otherwise, throw an exception.
   - 3.2. Substitute all associated collections of this instance with persistent collections.
   - 3.3. Attach the detached instance to the current session: cache entity, and add entity-entry, inside which the status is `Managed` and the loaded data copy remains empty, since at this point, the session doesn't know what is the original data loaded from the database.
   - 3.4. Perform the delete action on the instance (step 4).
4. Perform the delete action
   - 4.1. Modify the corresponding entity entry in the session cache: set the status as `Deleted`, and cache a deleted data copy, which is constructed by copying values from the instance but leaving option-type single associated references as empty.
   - 4.2. Invoke delete function on collection elements that have their cascade property set to `all` or `delete`.
   - 4.3. Add new delete event for the deleted data copy to the session's operation list.

     Note, that the delete action occurs on the deleted data copy, inside which all option-type associated references are set as empty. During the flushing process, an update event is invoked before the delete action to synchronise the table record to the values of the deleted data copy; this update breaks the potential cyclic relations.
   - 4.4. Invoke delete function on the reference entities that have their cascade property set to `all` or `delete`.

Figure 3.16: Cascading Delete Algorithm

1. Set flushing flag in the session, in order to avoid simultaneous multiple flushing processes.
2. Prepare flushing
   2.1. Prepare entity flushing: iterate over persistent entities to discover and save any newly referenced entity, also apply delete actions for orphan elements that are removed from collections that have the `delete_orphan` property.
   2.2. Prepare collection flushing: initialise the operation flags of collection entries, which indicate which kinds of operations (update, remove, or recreate) would be executed to maintain the relationship.
   2.3. Flush entities: detect dirty entities and schedule update event.
      a. If the entity is marked as `Deleted`, the dirty checking is performed by comparing its deleted data copy with its loaded data copy.
      b. Otherwise, comparing the content of the entity with its loaded data copy.
   2.4. Flush collections: determine collection actions.
      a. Recreate action, if the role of the collection has been changed, i.e. the collection is moved from one entity to another one.
      b. Update action, if the collection has newly added or removed elements.
      c. Remove action, if the collection is no longer referred to by entities.
3. Perform queued database operations by issuing SQL queries in the following order:
   3.1. Entity insert actions: insert new records into the table.
   3.2. Entity update actions: update data records in the table.
   3.3. Collection remove actions: clear foreign keys for collection elements.
   3.4. Collection update actions: update foreign keys for collection elements, including setting constraints for newly added elements, removing constraints for deleted elements, and updating constraints for existing elements.
   3.5. Collection recreate actions: change foreign keys for collection elements.
   3.6. Entity delete actions: delete corresponding records from the table.
4. Post flushing action:
   4.1. Remove entity entries for all deleted entities.
   4.2. Synchronise the loaded data copy for other entity entries.
5. Clear the flushing flag.

Figure 3.17: Session Flushing Process

Chapter 4

Embedded object-oriented query language

This chapter is an exploration of the possibilities of embedding a domain-specific querying language into a host language in a manner that is consistent with the object-relational mapping framework and thus enables us to perform set-oriented querying based on classes, objects and fields, while still maintaining compile-time type safety.

The embedded query language embraces the functionality of SQL, and also provides advanced features, beyond the expressiveness of traditional SQL, such as implicit join and nested query results, enabling to write shorter and more intuitive queries. By using the preprocessing technique, the query language is embedded in the host programming language in a syntax that is similar to standard SQL. Moreover, it leverages type inference to support parametrised and composable queries.

The major portion of this chapter, however, discusses how to guarantee the type safety of the embedded query language at compile time. The central idea is to use the OCaml compiler, not to directly check that the QQL query is well-typed, but rather to check that a type structure generated from the QQL query is well typed, where the relationship between this type structure and the QQL query is such that one is well typed if and only if the other is. The advantage of this approach is that, while the QQL type system does not lend itself to compile-time type checking by OCaml, the associated type structure does. We therefore propose a new idea of a *type avatar*, a dummy data structure that is generated to accompany each query to extend compile-time type checking to the query, and employ techniques, including phantom types [LM99a, CH03, FP06] to guarantee the type safety of generic aggregate functions, and use a variant of a generalised Hindley-Milner algorithm [Mil78] to discover type constraints for variables in queries. These type constraints are used to inform the type avatar generation. Finally, to execute the query against relational databases, we explain how our object-oriented queries can be translated into equivalent SQL.

4.1 Type avatars for domain specific sub-language type safety

Domain specific languages (DSLs), in contrast with general purpose languages, are designed to target particular problem domains. Such a language is called from programs written in general purpose languages, or directly embedded in the host language. Domain specific languages are typically designed with type systems that are different from that of the host language. Thus, an additional or modified type checking system is required to type check the full program across both the host language and the embedded domain specific language parts. Since providing such an extended type checking system often requires considerable extra work, it is not uncommon to either restrict the DSL type system to a subset of the host language type system — thus restricting the design of the DSL — or simply not to provide type checking across the interface between the host language and the DSL.

In order to provide a full solution to this inter-language type checking problem, we introduce

the new concept of *type avatars*, a practical technique to leverage the host language type checking mechanism to enable type checking across the host language and embedded domain specific language. A type avatar is a dummy data structure written in the host language that is generated before or at compilation time to model type constraints of code written in the DSL. This allows standard host language type checking to check not just the types used in the embedded DSL code, but also to check that those types are compatible with the types used in the host language that invokes, passes values to and receives return values from the embedded DSL code. The reference to 'dummy' means that avatar data is never executed or evaluated at any time; it is merely type checked at compile time.

For concreteness, we show a very simple example of a type avatar corresponding to a statement written in a putative natural language style DSL for use within the OCaml host language. Based on the above description, avatar data for the statement

```
tony's age is over 20
```

could be generated as (written as OCaml code),

```
module Avatar : sig end = struct
  let avatar () = tony.age > 20
end
```

As seen in the code, module `Avatar` is defined with an empty signature, thus avoiding exposing any local functions to the public. The DSL statement is transformed into the standard OCaml expression `tony.age > 20`, which accurately captures the original type constraints, namely that record `tony` has property `age` that is comparable and of the same type with `20`, and allows standard type checking.

However, it is not uncommon that type systems of DSLs cannot match the host language, and that therefore expressions written in such a DSL cannot be simply translated into type-constraint equivalent expressions in the host language. In such cases, the design and generation of suitable avatars can be complex. In the following sections, we demonstrate the design of an embedded compile-time type-safe object-oriented query language in OCaml by employing the technique of *type avatars* in combination with *phantom types* [LM99a, CH03, FP06] that allow us to correctly capture the types of our chosen query DSL in OCaml.

4.2 Qanat Query Language

4.2.1 QQL introduction

SQL is the standard query language for relational databases. However, because of the impedance mismatch problem, SQL does not match well with the programming models in modern popular programming languages, especially in an object-oriented context, where concepts of class, inheritance and composition, are foreign to the first normal form assumption of the relational data model. To mitigate this problem, many object-oriented database query languages have been proposed [ASL89, KR90, KKM93, BK06]. These object-oriented query languages are either used directly in object-oriented databases (e.g. OQL [ASL89]), or incorporated in ORM frameworks (e.g. HQL [BK06], JDOQL [TVM+03]) for execution on relational databases. Instead of using tables and columns, object-oriented queries allow us to define queries in a much shorter and more intuitive approach, based on the programming model. However, one of the negative aspects of these approaches is that queries, and their results, are represented as strings in the program and are not checked at compile time by the type system of the programming language.

Our approach is a pragmatic one, starting by embedding the query directly in the program code. It has similarities to the approach of SQLJ in that the program is pre-processed, then passed to the compiler. In practice, the embedded queries are extracted and parsed into *abstract syntax trees* (ASTs), which are verified and transformed into standard code. During the transformation process, additional actions, such as code generation and type checking, are performed. Finally, the transformed AST is directly fed into the compiler for byte code generation. Our approach differs from SQLJ in that our query language is object-oriented, and in the nature of the compile-time type safety guarantees the respective systems can make. We call this query language the *Qanat*

Query Language (QQL), after the name of our proposed ORM framework, Qanat. QQL adopts a syntax inspired by Hibernate's HQL [BK04, BK06].

Each query in QQL starts with a qualified module name that provides the object-relational mapping facility, followed by a query expression, wrapped in a pair of squared brackets. The following code demonstrates the core query expression (also one of the simplest queries):

```
Session.[from customer]
```

This retrieves all entities of class `customer`. The select clause is optional in QQL, and can be inferred from the query expression (i.e. the from part).

Queries in QQL can be defined with parameters, and in a navigational style following the entity references. The elements in the query expression are OCaml objects and object field expressions, rather than SQL tables, records or columns. The results returned will be lists of OCaml objects or lists of OCaml tuples of objects. Thus we stay in the OCaml language model rather than switching to an SQL model. For instance, one can obtain the number of placed orders for customers whose name ends with a specified string as follows:

```
let q = Session.[select cust#email, cust#orders#count()
                 from customer as cust
                 where cust#name like :vname]
in
(* execute the query with a specified parameter *)
Session.query sess (q ~vname:"%tom")
```

Such a query in QQL is often shorter and easy-to-write than the equivalent SQL, shown below, which involves a nested query and explicit join conditions. That is because QQL supports defining queries in a navigational way, without explicitly writing table join and corresponding ojoin conditions.

```
select t1.email,
  (select count (t3.orderid)
   from customer t2 left join order t3 on t2.custid = t3.custid
   where t2.custid = t1.custid)
from customer t1
where t1.name like "%tom"
```

Figure 4.1 shows a subset of the QQL grammar; the formal and complete query language is defined in Chapter 6. Symbol $[\ ]^{0.1}$ indicates an optional clause or expression. By default, when `select` is not specified, it returns all entities involved in the query, i.e. those referred to in the `from` clause. The `group-by` clause differs slightly from SQL in that, in addition to specifying the criteria for group allocation, it allows an alias for each specified item, which could be referred to in the `select` as one of the returned value. Other clauses are the same as the ones in SQL, such as `where` and `having`, which define boolean predicates for filtering results, `order-by`, which specifies the sorting strategy, and `limit` and `offset`, which set the number of returned rows and the start position.

QQL has some differences from, and advantages over, SQL. Instead of supporting generic operators, QQL provides unique operators for various type values in order to be consistent with the OCaml language, such as addition operator `(+)` for int values, and `(+.)` for float values[1]. As an object-oriented language, it allows expressing query conditions navigationally by following entity references, and returning nested results. In addition, it supports incorporating variables in queries, and composing queries as components. More importantly, with the aid of type avatars, these queries can be checked for type safety at compile-time by leveraging OCaml's type checker.

Each query written in QQL is compiled into a query object that encapsulates expression trees representing the query, avatar data for type checking, and utility data for query composition and result construction. Queries with variables are compiled into functions with parameters for generating the query object. Then these queries can be executed against a session instance, which invokes variable binding, SQL generation, execution and result set construction. Alternatively, they can be composed with other queries to construct complex queries.

[1]The latter has been omitted from the grammar extract in Figure 4.1 but appears in the full grammar in Chapter 6.

QQL query

q ::=

[select *s*]$^{0.1}$ from *f* [where *e*]$^{0.1}$
[group by *g* [having *e*]$^{0.1}$]$^{0.1}$
[order by *o*]$^{0.1}$
[limit *ie*]$^{0.1}$ [offset *ie*]$^{0.1}$

From expression

f ::=

ae	field access expr
| :*v*	variable
| *ae* as id	renaming
| *f*, *f*	cartesian product
| *f* join *f* on *ae* = *ae*	explicit join

Select expression

s ::=

c	constant value
| *ae*	field access expr
| fn *ae*	aggregate function (fn=count,max,min,avg,sum)
| *ae*#fn ()	aggregate function on collection-type field
| *s* as id	renaming
| *s*, *s*	multiple select exprs

GroupBy expression

g ::=

ae as *id*	groupby item
| *g*, *g*	multiple group exprs

OrderBy expression

o ::=

ae [asc |desc]$^{0.1}$	default direction (asc)
| *o*, *o*	multiple order exprs

Field access expression

ae ::=

id	identifier
| *ae*#*id*	object field access
| *ae*.*id*	record field access
| *ae*#[id => *ae*]	collection field mapping
| valof *ae*	value of option type
| itemsof *ae*	items of a collection

Expression

e ::=

c	constant value
| :*v*	variable
| *ae*	field access expr
| *e*,*e*	pair of expression
| [*e*;...;*e*]	collection
| (*e*)	expr with bracket
| fn *ae*	aggregate function (fn=count,max,min,avg,sum)
| *e op e*	infix operator (op=+,-,*,/,and,or,in,...)
| *e* is [not]$^{0.1}$ Null	is Null checking
| *e* between *e* and *e*	between operator
| *ae*#fn ()	aggregate function on collection-type field

Limit/Offset expression

ie ::=

i	integer
| :*v*	variable

Figure 4.1: QQL Grammar

The remainder of this chapter is organised as follows. In section 4.2.2 we present and discuss the QQL type system. In section 4.3, we describe how we use phantom types and predefined shadow functions to provide type checking in OCaml on SQL's overloaded functions. In section 4.5, we discuss type avatars and their role in achieving type checking on QQL queries. In the following section, 4.7, we propose a simplified type inference algorithm to incorporate variables in our type checking of QQL queries. In the last two sections 4.8 and 4.9, we add support for query composition to QQL, and, finally, demonstrate how to translate the object-oriented QQL into equivalent relational SQL.

4.2.2 The type system of QQL

The type system of QQL is much simpler than those of traditional programming languages . Although it has predefined functions (e.g. `sum`, `valof`, etc), QQL does not support user-defined functions. Although it has an sub-typing structure that forms an acyclic graph, rather than the more common purely hierarchical arrangement, all types that are not at the leaf level of the type

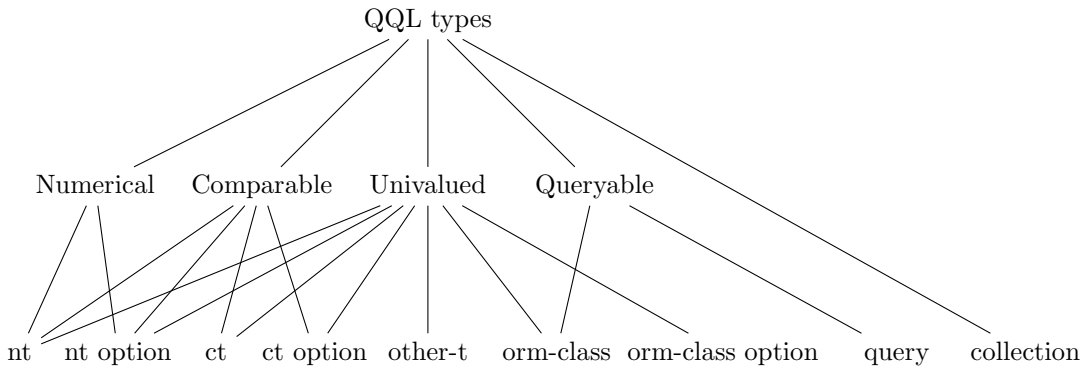
graph are predefined and abstract(c.f. Figure 4.2). The QQL type system consists of *basic types* that are constructed based on primitive types (e.g. `int`, `string`, etc) either in the form of *option*, *tuple*, *record*, *anonymous-object* or *variant*, *orm types* that are defined in the orm module and have object-relational mapping facilities provided by the `Session` module, and *collection* and *query* types. *Collection* values can be constructed either using the constructor `[x1; x2; ...; xn]` or using the collection field mapping `ae#[x => e]`, where `ae` is an orm field access expression (i.e. an expression indicating an reference chain beginning at an orm object). *Query* is defined using a *select-from-where* style expression such as `Session.[select s from f as x where e]`. Let *qql-type* be the set of QQL types that can be represented as:

$$
\begin{array}{lcll}
\textit{qql-type} & = & \textit{t} \mid \textit{(t collection) query} & \\
\\
\textit{t} & = & \textit{basic-type} \mid \textit{orm-type} \mid \textit{(t) collection} & \\
\\
\textit{basic-type} & = & \textit{primitive-type} & \\
 & \mid & \textit{primitive-type option} & \textit{(* option type *)} \\
 & \mid & \textit{basic-type}_1 \times \cdots \times \textit{basic-type}_n & \textit{(* tuple type *)} \\
 & \mid & \{r_1 : \textit{basic-type}_1; \cdots ; r_n : \textit{basic-type}_n\} & \textit{(* record type *)} \\
 & \mid & \langle m_1 : \textit{basic-type}_1; \cdots ; m_n : \textit{basic-type}_n \rangle & \textit{(* anonymous object type *)} \\
 & \mid & \left[C_1 \texttt{ of } \textit{basic-type}_1^1 \times \cdots \times \textit{basic-type}_1^j | \cdots | C_n \texttt{ of } \textit{basic-type}_n^1 \times \cdots \times \textit{basic-type}_n^k\right] & \\
 & & & \textit{(* variant type *)} \\
\\
\textit{primitive-type} & = & \textit{int, float, char, bool, } \ldots & \\
\\
\textit{orm-type} & = & \textit{orm } \langle m_1 : t_1; \cdots ; m_n : t_n \rangle \mid \textit{(orm } \langle m_1 : t_1; \cdots ; m_n : t_n \rangle \textit{) option} &
\end{array}
$$

Types in QQL are grouped into an acyclic sub-typing graph (c.f. Figure 4.2), and can be described as follows:

$$
\begin{array}{ll}
\textit{Numerical} & = \textit{int} \mid \textit{int option} \mid \textit{float} \mid \textit{float option} \mid \cdots \mid \textit{Numeric.t} \mid \textit{Numeric.t option} \\
\\
\textit{Comparable} & = \textit{char} \mid \textit{char option} \mid \textit{int} \mid \textit{int option} \mid \cdots \mid \textit{Numeric.t} \mid \textit{Numeric.t option} \\
\\
\textit{Queryable} & = \textit{orm } \langle m_1 : t_1; \cdots ; m_n : t_n \rangle \mid \textit{(t collection) query} \\
\\
\textit{Univalued} & = \textit{basic-type} \mid \textit{orm } \langle m_1 : t_1; \cdots ; m_n : t_n \rangle \mid \textit{orm } \langle m_1 : t_1; \cdots ; m_n : t_n \rangle \textit{ option} \\
\\
\textit{QQLTypes} & = \textit{Numerical} \mid \textit{Comparable} \mid \textit{Queryable} \mid \textit{Univalued} \mid \textit{(t) collection}
\end{array}
$$

The abstract elements of this graph structure are *QQLTypes*, as the root of the graph, and *Numerical*, *Comparable*, *Queryable* and *Univalued* (c.f. Figure 4.2). We write $\tau <:$ C to denote that τ is a subtype of C. Values of *Numerical* type can be used in overloaded functions that perform arithmetic calculation, such as `sum` and `avg`. Values of *Comparable* type can be used in functions that require an order comparison, such as `min` and `max`, as well as in expressions involving comparison operators. The `order by` clause, also requires parameters of type *Comparable*. Values of *Queryable* type (including *orm class* and *query*) are required in the *from* clause of a query. Values of *Univalued* type (in contrast with *collection* types) are required define the `group-by` partition criteria (it would not make sense to try to group a set of tuples by a value of type collection). Collection types are not subtypes to any of the non-root types. Values of this type can be used as query return values (i.e. in the `select` clause) or used in field-access-expression based aggregate methods, e.g. `max` can be invoked on a *Comparable* collection (i.e. where the elements of the collection are of *Comparable* type). Query and collection types are polymorphic types; the type variable indicates the type of the query results and the type of the encapsulated elements, respectively.



`nt`	$= int \mid float \mid int32 \mid \cdots \mid Numeric.t$
`ct`	$= char \mid string \mid \cdots \mid timetz$ (time with time-zone)
other-t	$= bool \mid bool\ option \mid$ record-type $\mid$ tuple-type $\mid$ variant-type $\mid$ anonymous-object
orm-class	$= orm\ < m_1 : t_1; \cdots ; m_n : t_n >$

Figure 4.2: QQL Types

4.2.3 QQL typing rules

We closely model our approach in typing a QQL query on Bierman and Trigoni's approach to typing ODMG OQL [BT00]. Although there are considerable differences in the languages being typed, much of the general structure and flavour of the typing rules carries over very well to QQL, even if details differ.

To type a QQL query, we assume that we have constructed a schema typing environment, noted as S, which contains all orm classes predefined in the orm module and the structure of each orm class. Based on this typing environment, QQL allows us to use orm class types in the `from` clause. This S is a partial function mapping class names to their type information (including attributes and relations). More formally

$$S : Id \rightharpoonup \text{orm-type}$$

Unlike in [BT00], we do not need to keep the method type information of orm classes in S, current version of QQL doesn't allow to use or refer to orm class methods or functions in the query. The support of mapping user defined class methods and functions to underying database functions or procedures in a compile-time type-safe way remains an open problem for the future research.

Another typing environment D contains the variable definitions that can be passed as sub queries (i.e. query type variables in the `from` clause) to another query, or used as values of other QQL types in the `where`, `having`, `limit` and `offset` clauses.

$$D : Id \rightharpoonup \text{qql-type}$$

We thus need three typing environments for the schema S, variable definitions D and typing environment Γ, where Γ contains the types of free identifiers in `q`. As is standard practice, we use the shorthand notation $\Gamma, x : \tau$, to denote an extended typing environment $\Gamma \cup \{x : \tau\}$. Thus the typing judgement for a query is written as

$$\mathrm{S}; \mathrm{D}; \Gamma \vdash \texttt{q} : \textit{'a query}$$

We assume a scheme of axioms for literals of the following form, where i is an integer literal and s is a string literal etc.

$$\frac{}{\mathrm{S}; \mathrm{D}; \Gamma \vdash \texttt{i} : int}\ \textsc{Lit-int} \qquad \frac{}{\mathrm{S}; \mathrm{D}; \Gamma \vdash \texttt{s} : string}\ \textsc{Lit-string} \qquad \ldots$$

We obtain types for QQL variables from the D environment.

$$\frac{}{\mathrm{S};\mathrm{D},\mathtt{v}:\tau;\Gamma\vdash \mathtt{:v}:\tau}\ \textsc{Var}$$

Next are the typing rules for path expressions, including object field access, record field access, collection field mapping, `valof` and `itemsof`, and tuples.

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{x}:<m_1:t_1;\cdots;m_n:t_n> \qquad \mathtt{1}\le\mathtt{i}\le\mathtt{n}}{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{x\#}m_i:t_i}\ \textsc{Object field access}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{x}:\{r_1:t_1;\cdots;r_n:t_n\} \qquad \mathtt{1}\le\mathtt{i}\le\mathtt{n}}{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{x.}r_i:t_i}\ \textsc{record field access}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{vs}:\tau\ \textit{collection} \qquad \mathrm{S};\mathrm{D};\Gamma,\mathrm{x}:\tau\vdash \mathtt{e}:\sigma}{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{vs\#[x => e]}:\sigma\ \textit{collection}}\ \textsc{collection field mapping}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{e}:\tau\ \textit{option}}{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{valof\ e}:\tau}\ \textsc{Valof} \qquad \frac{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{e}:\tau\ \textit{collection}}{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{itemsof\ e}:\tau}\ \textsc{Itemsof}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash x_1:t_1 \qquad \cdots \qquad \mathrm{S};\mathrm{D};\Gamma\vdash x_n:t_n}{\mathrm{S};\mathrm{D};\Gamma\vdash (x_1,\cdots,x_n):t_i\times\cdots\times t_n}\ \textsc{tuple}$$

Next we give typing rules for query expressions. There are a few points to note here:

- The $\textsc{Arithmetic}_{op}$ is a scheme of rules, one for each arithmetic binary operator in the QQL grammar. These include "`+`", for addition of `int` values, "`*.`" for multiplication of `float` values, "`+/`" for addition of `numeric.t` (a number type suitable for currency values) and others. Note that there are no such operators for `option` types (e.g. *int option*); the `valof` operation would have to be applied to such values first. This approach corresponds to arithmetic operators in OCaml.
- The $\textsc{Comparison}_{op}$ is also a scheme of rules, one for each comparison operator, "`=, <, >, <=, >=, <>, !=`". Unlike the arithmetic operators, we do not require different comparison operators for the same logical comparison on different types, and the operators work on `option` types as well. This also corresponds to comparison operators in OCaml.
- Some rules, i.e. those for aggregate functions, Between-and, In, and for Comparison are defined with constraints, indicating that these functions can only be applied to proper type values. QQL supports two kinds of aggregate expressions, one is based on the `group-by` clause (similar to standard SQL), and another is based on collection-type field expressions. It should be noted that the two forms of each aggregate function take different types to operate on. For example, the field expression form of `sum`, as shown in Sum-col, acts on a *Numerical collection* to return a *Numerical* of the same type as appeared in the collection. The SQL style form (Sum), however, is surprising as it appears to act on a *Numerical*, instead of a *Numerical collection*. In fact, the underlying summation function will be applied to a collection values of the *Numerical* type indicated inside the parentheses following the `sum` keyword, but the syntax, following SQL, requires that the contents of those parentheses identify the field to be extracted from each tuple of the result partition for summing rather than requiring a collection of such values.

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \tau \qquad \mathit{arith}_{op} <: \tau \to \tau \to \tau \qquad \tau <: \mathit{Numerical}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \ \texttt{arith}_{op}\ e_2 : \tau}\ \textsc{Arithmetic}_{op}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \tau \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \tau \qquad \tau <: \mathit{Comparable}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \ \texttt{compare}_{op}\ e_2 : \mathit{bool}}\ \textsc{Comparison}_{op}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \mathit{bool} \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \mathit{bool}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \ \texttt{and}\ e_2 : \mathit{bool}}\ \textsc{And}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \mathit{bool} \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \mathit{bool}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \ \texttt{or}\ e_2 : \mathit{bool}}\ \textsc{Or} \qquad \frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \mathit{bool}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{not e} : \mathit{bool}}\ \textsc{Not}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau \qquad \tau <: \mathit{Numerical}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{sum e} : \tau}\ \textsc{Sum} \qquad \frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau \qquad \tau <: \mathit{Numerical}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{avg e} : \tau}\ \textsc{Avg}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau \qquad \tau <: \mathit{Comparable}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{max e} : \tau}\ \textsc{Max} \qquad \frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau \qquad \tau <: \mathit{Comparable}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{min e} : \tau}\ \textsc{Min}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \alpha}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{count e} : \mathit{int}}\ \textsc{Count} \qquad \frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \alpha\ \mathit{collection}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e\#count()} : \mathit{int}}\ \textsc{Count-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau\ \mathit{collection} \qquad \tau <: \mathit{Numerical}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e\#sum()} : \tau}\ \textsc{Sum-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau\ \mathit{collection} \qquad \tau <: \mathit{Numerical}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e\#avg()} : \tau}\ \textsc{Avg-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau\ \mathit{collection} \qquad \tau <: \mathit{Comparable}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e\#max()} : \tau}\ \textsc{Max-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e} : \tau\ \mathit{collection} \qquad \tau <: \mathit{Comparable}}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{e\#min()} : \tau}\ \textsc{Min-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \tau \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \tau \qquad \cdots}{\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{[}e_1\texttt{; }e_2\texttt{; }\cdots\texttt{]} : \tau\ \mathit{collection}}\ \textsc{List}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \tau \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \tau\ \mathit{collection} \qquad \tau <: \mathit{Univalued}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \ \texttt{in}\ e_2 : \mathit{bool}}\ \textsc{In}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \mathit{string} \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \mathit{string}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \ \texttt{like}\ e_2 : \mathit{bool}}\ \textsc{Like}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 : \tau \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_2 : \tau \qquad \mathrm{S};\mathrm{D};\Gamma \vdash e_3 : \tau \qquad \tau <: \mathit{Univalued}}{\mathrm{S};\mathrm{D};\Gamma \vdash e_1 \texttt{ between } e_2 \texttt{ and } e_3 : \mathit{bool}} \textsc{Between-and}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash e : \tau \ \mathit{option} \qquad \tau <: \mathit{Univalued}}{\mathrm{S};\mathrm{D};\Gamma \vdash e \texttt{ is [not]}^{0.1} \texttt{ Null} : \mathit{bool}} \textsc{Is-null}$$

Finally, we show the typing rules for the `select-from-where`, SFW, and the `select-from-where-groupby-having`, SFWGH query forms.

A number of manipulations of the query are performed before we consider typing issues:

- Even though the `select` clause in QQL is optional, in the case that it is omitted, QQL infers a default `select` clause, i.e. returning all entities referred to in the `from` clause, which is similar to SQL's `select * from tables`.
- Similarly, for a missing optional `where` or `having` clause, a default version with a `true` condition is inferred.
- Aliases can optionally be defined for any element in the `from` (or the `group by`) clause. If any such alias is omitted, an alias is generated for it, thus we need not write typing rules that cover such omissions.
- Any occurrence of an explicit `join` expression in the `from` clause is re-written as a sequence of orm and/or query variable expressions in the `from` clause and the appropriate modification of the `where` clause. Hence this also does not need to be directly addressed in the typing rules.
- If an SQL style aggregate function application appears in the query, but there is no `group by` clause, then a default `group by` clause will be inferred the groups all the results into one partition. This allows us to avoid writing separate rules for the different, but equivalent, cases.

Consider the rule `SFW`, reading from the top down. First we have the judgement that variable expression : p_1 is of type (ε_1 *collection*) *query*, p_1 must be declared as a query-type variable in environment D (as witnessed by the rule VAR). Thus query identifier v_1 is of type ε_1, representing the type of the sub-query return values, and may occur free in the following expressions. The next few judgements are for query-type variables from : p_2 to : p_k. We then form judgements q_i which is of type σ_i *collection*, where σ_i are the types of orm classes which appear in the schema environment S. In fact, the ordering of query variables and orm class expressions can be arbitrary but we show it in this fixed order of variables first and then non variable expressions mainly because it significantly simplifies the presentation, but also because any ordering of variables and orm class expressions in the from clause is semantically equivalent to an ordering with variables first (because variables can be used in orm expressions but not vice versa).

$\bar{x}$ represents a sequence x_i. The judgement `w` constructs a boolean predicate, with free identifiers from $\bar{v}$, $\bar{x}$ and Γ, for the `where` clause. We form the last judgement that expression s is of type τ based on typing environment Γ. Given these judgements, we conclude that the select query is of a function type, from various sub-query types (ε_i *collection*) *query* to query type (τ *collection*) *query*. In the case that there is no sub-query variables, the select query is of query type directly.

$$\frac{\begin{array}{l}
\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{:p}_1 : (\varepsilon_1 \textit{ collection}) \textit{ query} \\
\mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_1 : \varepsilon_1 \vdash \texttt{:p}_2 : (\varepsilon_2 \textit{ collection}) \textit{ query} \\
\quad\vdots \\
\mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_{k-1} : \varepsilon_{k-1} \vdash \texttt{:p}_k : (\varepsilon_k \textit{ collection}) \textit{ query} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon} \vdash \texttt{q}_1 : \sigma_1 \textit{ collection} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1 \vdash \texttt{q}_2 : \sigma_2 \textit{ collection} \\
\quad\vdots \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1, \ldots, \mathrm{x}_{n-1} : \sigma_{n-1} \vdash \texttt{q}_n : \sigma_n \textit{ collection} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash \texttt{w} : \textit{bool} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash \texttt{s} : \tau
\end{array}}{\begin{array}{ll}
\texttt{select s from :p}_1 \texttt{ as v}_1 \texttt{ ... , :p}_k \texttt{ as v}_k & (\varepsilon_1 \textit{ collection}) \textit{ query} \to \cdots \to \\
\mathrm{S};\mathrm{D};\Gamma \vdash \quad \texttt{q}_1 \texttt{ as x}_1\texttt{, ... , q}_n \texttt{ as x}_n\texttt{,} \quad : & (\varepsilon_k \textit{ collection}) \textit{ query} \to \\
\texttt{where w} & (\tau \textit{ collection}) \textit{ query}
\end{array}} \text{ SFW}$$

The `SFWGH` rule is similar to the typing rule `SFW`, except that `SFWGH` constructs judgements for `group-by` and `having` clauses. Judgements for g_1 to g_j bind identifiers k_1 to k_j in Γ. Each of these k_i is of type τ_i, where τ_i must be *Univalued*. Then judgements for h and s can be constructed based on $\bar{v}$, $\bar{x}$, $\bar{k}$ and identifiers from Γ. As in SQL, queries in QQL can only return values from fields/entities that are referred to in the `group by` clause in the case that `group-by` is defined or group-by based aggregate functions are used in the query. Such constraints are guaranteed by the QQL grammar, rather than the type system.

$$\frac{\begin{array}{l}
\mathrm{S};\mathrm{D};\Gamma \vdash \texttt{:p}_1 : (\varepsilon_1 \textit{ collection}) \textit{ query} \\
\mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_1 : \varepsilon_1 \vdash \texttt{:p}_2 : (\varepsilon_2 \textit{ collection}) \textit{ query} \\
\quad\vdots \\
\mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_{k-1} : \varepsilon_{k-1} \vdash \texttt{:p}_k : (\varepsilon_k \textit{ collection}) \textit{ query} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon} \vdash \texttt{q}_1 : \sigma_1 \textit{ collection} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1 \vdash \texttt{q}_2 : \sigma_2 \textit{ collection} \\
\quad\vdots \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1, \ldots, \mathrm{x}_{n-1} : \sigma_{n-1} \vdash \texttt{q}_n : \sigma_n \textit{ collection} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash \texttt{w} : \textit{bool} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash \texttt{g}_1 : \tau_1 \qquad \tau_1 <: \textit{Univalued} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \mathrm{k}_1 : \tau_1 \vdash \texttt{g}_2 : \tau_2 \qquad \tau_2 <: \textit{Univalued} \\
\quad\vdots \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \bar{\mathrm{k}} : \bar{\tau}, \mathrm{k}_1 : \tau_1, \ldots, \mathrm{k}_{j-1} : \tau_{j-1} \vdash \texttt{g}_j : \tau_j \qquad \tau_j <: \textit{Univalued} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \bar{\mathrm{k}} : \bar{\tau} \vdash \texttt{h} : \textit{bool} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \bar{\mathrm{k}} : \bar{\tau} \vdash \texttt{s} : \varphi
\end{array}}{\begin{array}{ll}
\texttt{select s from :p}_1 \texttt{ as v}_1 \texttt{ ... , :p}_k \texttt{ as v}_k & \\
\texttt{q}_1 \texttt{ as x}_1\texttt{, ... , q}_n \texttt{ as x}_n\texttt{,} & (\varepsilon_1 \textit{ collection}) \textit{ query} \to \cdots \to \\
\mathrm{S};\mathrm{D};\Gamma \vdash \quad \texttt{where w} \quad : & (\varepsilon_k \textit{ collection}) \textit{ query} \to \\
\texttt{group by g}_1 \texttt{ as k}_1\texttt{,...,g}_j \texttt{ as k}_j & (\varphi \textit{ collection}) \textit{ query} \\
\texttt{having h} &
\end{array}} \text{ SFWGH}$$

We do not present rules for the variations of SFW and SFWGH that include the `distinct` keyword, or the `order by`, `limit`, or `offset` clauses. Considering the combinations possible, doing so would be straightforward but long and tedious, and not very enlightening for the reader. In fact, the addition of none of these clauses cause any change to the return type of a query; the collection that a query returns will always be a list of some type τ, even if duplicates have been removed by the `distinct` keyword or whether or not the results have been ordered. Thus the change to the type rules would merely require that the expressions in the `limit` and `offset` clauses be of type

int and derivable from the environments, while the expressions listed in the `order by` clause must each be of type *Comparable* and derivable from the environments.

We end this section with a substitution lemma, to show, as usual, that in our type system appropriate subtstitution of matching expressions for variables preserves types.

Lemma 4.1 (Preservation of types under Substitution) *If* $\mathrm{S;D;\Gamma,x:T} \vdash \texttt{q} : Q$ *and* $\mathrm{S;D;\Gamma} \vdash \texttt{y} : T$*, then* $\mathrm{S;D;\Gamma} \vdash \texttt{[x} \mapsto \texttt{y]q} : Q$.

Proof: By induction on a derivation of the statement $\mathrm{S;D;\Gamma,x:T} \vdash \texttt{q} : Q$. for a given derivation, we proceed by cases on the final typing rule used in the proof. Since lambda abstraction and user-defined functions are not allowed in QQL, the proof of the substitution lemma is much simpler than is commonly the case in other languages.

Case - VAR: $q = z$ with $z : Q \in (\Gamma, x : T)$

There are two sub-cases to consider, depending on whether z is x or another variable. If $z = x$, then $[x \mapsto y]z = y$. The required result is then $\mathrm{S;D;\Gamma} \vdash \texttt{y} : T$, which is among the assumptions of the lemma. Otherwise, $[x \mapsto y]z = z$, and the desired result is immediate.

Case - LIT-INT: Suppose $q = n$ where n is an integer, $Q = int$

Then $[x \mapsto y]q = n$, and the desired result, $\mathrm{S;D;\Gamma} \vdash \texttt{[x} \mapsto \texttt{y]q} : Q$, is immediate.

All the remaining cases are variants of applications of built in functions or operations, since user-defined functions are not supported. Therefore, to complete the lemma we need only show the cases for each of these applications. As all of these are very similar, we show here only one of these cases; namely, that for arithmetic operations.

Case - $\mathrm{ARITHMETIC}_{op}$: $q = e_1 \texttt{arith}_{op} e_2$, $\mathrm{S;D;\Gamma} \vdash e_1 : \tau$, $\mathrm{S;D;\Gamma} \vdash e_2 : \tau$, $Q = \tau$

The use of the induction hypothesis yield $\mathrm{S;D;\Gamma} \vdash [x \mapsto y]e_1 : \tau$, and $\mathrm{S;D;\Gamma} \vdash [x \mapsto y]e_2 : \tau$, from which the result follows by the $\texttt{ARITHMETIC}_{op}$ rule.

4.3 Phantom types and shadow functions

SQL, and therefore QQL, provide built-in aggregate functions that are executed on the database server. These functions include, among others, `sum`, `count`, `avg`, `max`. They are overloaded to work on various types. Thus `sum` can be applied to a column of integers to return an integer, a column of floats to return a float, etc. OCaml, on the other hand, does not allow overloaded functions. Therefore, when parsing QQL queries, we must ensure that the way the aggregate function is invoked in the query is type compatible with the orm classes it is applied to, the remainder of the QQL query, and the OCaml variables that the return values are assigned or bound to. We do this by predefining a *shadow function* for each QQL aggregate function and using phantom types to encode the type constraints on them. A shadow function is a dummy function whose parameters and return types match those of the built-in aggregate function. These shadow functions are never executed, but they appear in expressions in the code generated by the preprocessor so that a compile-time error will be triggered by these functions if and only if the built-in aggregate function is being used in a type incorrect way. For example, if a `sum` function is being applied to a column of floats, but the return value from that function is being bound to an integer variable. We will discuss shadow functions further, with examples, in Section 4.3.3.

Standard ML and OCaml use the Hindley-Milner type inference algorithm [Mil78] to guarantee compile-time type safety of programs. however, this algorithm does not support features like function overloading. Even though parametric polymorphism and type constructors can be used to capture program properties beyond the standard Hindley-Milner algorithm, it is still difficult to provide a direct solution to type-safe function overloading. The SQL language supports overloaded functions, such as `sum` and `avg` on various numeric types, `max` and `min` on comparable types, and arithmetic operators which are applied on identically typed numerical values. We refer to function overloading on the type of parameters, rather than any broader concept of function overloading.

Type constructors provide the ability to unify various types into a single type, known as union type in OCaml. For concreteness, we define union types, `num` and `cmp`, as follows (in OCaml syntax),

```
type num = Int of int | Float of float | Int32 of int32
type cmp = Int of int | Char of char | String of string
```

We define a number of operations performing calculation on these types in Figure 4.3. This raises the problem that a certain type can be categorised into more than one category, as type `int` can be in `num` and `cmp` simultaneously. Another problem is that such "well-typed" expressions can cause run-time exceptions. For example, the type system considers "`plus (Int 1) (Float 2.0)`" to be well-typed, but evaluating this expression raises a run-time type mismatch exception. We summarise the problem as the lack of an approach to distinguish each concrete subtype.

```
type num = Int of int | Float of float | Int32 of int32

let plus (a:num) (b:num) =
   match a, b with
  |(Int x), (Int y) -> Int (x + y)
  |(Float x), (Float y) -> Float (x +. y)
  |(Int32 x), (Int32 y) -> Int32 (Int32.add x y)
  |_, _ -> failwith "type mismatch"
```

Figure 4.3: Type unsafe operations on union types

4.3.1 Phantom types

The term *phantom type* was originally coined by Leijen and Meijer in Haskell/DB [LM99a] to denote typed arithmetic expressions. The basic idea of phantom types is to use free type parameters when constructing data types, but not use these type parameters in the right hand side of the type definition. The fact that the phantom parameter is unused on the right hand side gives the freedom to use them to encode additional information about types, which instructs the type system to enforce the difference of these values.

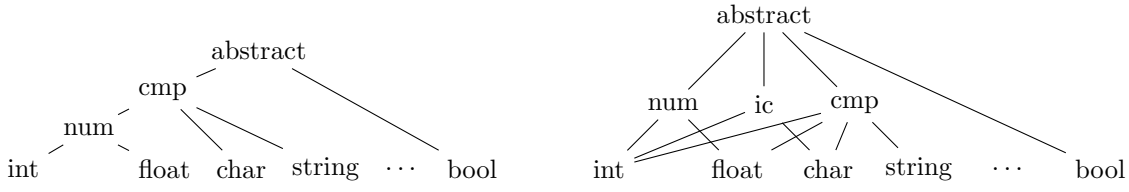
As an introductory example, we show how to use phantom types to encode the access control rights associated with simple read-only/read-write file access. Since the type variable is not used in the type definition, it is free at runtime. The way to enforce the type system to track the type variable is by constraining the phantom types using a signature.

In the following example, a file handler is opened in mode `readwrite`. It can be changed to mode `readonly` by applying function `change_to_readonly`.

```
type readonly
type readwrite

module File : sig
  type 'a t

  val open_file : string -> readwrite t
  val change_to_readonly : 'a t -> readonly t
  val read : 'a t -> string
  val write : readwrite t -> string -> unit

end = struct

  let open_file filename = open_file filename
  let change_to_readonly file = file
  let read file = read_text file
  let write file text = write_text file text
end
```

In the above code, the phantom type indicates the permission of the file handler. A file handler with phantom type `readwrite` can be read from or written to; while a read-only file handler can only be read from. Function `read` accepts any kind of file handler as parameter; `write` can only work on a read-write connection. Function `change_to_readonly` restricts the access rights of the file handler to `readonly`. Thus any attempt to write through a `readonly` file handler will trigger a compile-time error.

Phantom types [CH03, FP06] are often used for encoding bounds checks in the type system, or, in the case of function overloading, for distinguishing various sub-typings. Fluet and Pucella [FP06] investigate encodings for various sub-typing hierarchies. For nodes in a tree hierarchy sub-typing graph, phantom type encodings represent the path from the top of the tree to the node itself. For example, the sub-typing of `num` in Figure 4.4a could be represented as `(('a num) cmp) abstract`; the phantom encoding for `int` is `((int num) cmp) abstract`.



(a) Tree hierarchy (b) Overlapped multiple inheritances

Figure 4.4: Subtyping hierarchy

In the aggregate function case, the sub-typing hierarchy could be depicted as in Figure 4.4a, where `num` represents a *numeric* type, `cmp` represents a *comparable* type. Type `num` happens to be a subtype of `cmp`, thus it is suitable to use the tree hierarchy encoding [FP06]. However, we can not stop programmers declaring functions that overload on `int` and `char`, but without `float`, like the function `nextval`, included in the library of many databases. Such a function `nextval` takes an integer or character as input and return another integer or character respectively, obtained by increasing the integer value by one in the case of integers or returning the following character in the ASCII table in the case of characters. More generally, we need an encoding to represent overlapped multiple inheritance sub-typing hierarchies, such as the one depicted in Figure 4.4b, where `ic` represents a type that can be incremented.

We use two notations (borrowing from [FP06]) $\langle . \rangle_C$ and $\langle . \rangle_A$, for describing the phantom encoding. Suppose type σ exists in the sub-typing hierarchy. Then $\langle \sigma \rangle_C$, called the *concrete* encoding, represents the phantom encoding (i.e. the unused type variable) of σ, and $\langle \sigma \rangle_A$ represents the abstract encoding of σ, used to restrict the domain of an operation to values that are subtypes of σ. Thus functions which accept parameters that can be of a specific type or any subtype of that type should use the abstract encoding, whereas the type of any specific value, and hence the return type of any function, should use the concrete encoding.

The principle of encoding is to record, in the type variable, all parent type information. Let S be the finite set corresponding to the sub-typing hierarchy, and assume an ordering $s_1, \ldots, s_k, \ldots, s_n$ for elements (excluding the root element) of S. s_1 to s_k represent intermediate (non-leaf) nodes, s_{k+1} to s_n are leaf nodes. First, we define a data type that acts as a *boolean* value at the level of types,

```
type 'a z = Irrelevant_z
```

where `unit` and `unit z` represent *true* and *false* respectively. The encoding of sub-typing hierarchy S could be defined as:

```
type r_σ = RepresentType_σ
```

$\langle \sigma \rangle_C$ = $(t_1$ * $(t_2$ * $(t_3$ * ... * $(t_k$ * $r)...)))$

where t_i = $\begin{cases} unit & if\ s_i \in Path(\sigma) \\ unit\ z & otherwise \end{cases}$

and r represents type σ, i.e. r_σ

and k is the number of non-leaf elements (excluding the root element)
and $Path(\sigma)$ is the set of elements that locate on any path from the root to σ

$\langle\sigma\rangle_A$ = $(t_1 * (t_2 * (t_3 * \ldots * (t_k * r)\ldots)))$

where $t_i = \begin{cases} unit & if\ s_i = \sigma \\ 'a_i & otherwise \end{cases}$

and $r = \begin{cases} r_\sigma & if\ \sigma\ is\ leaf\ node \\ 'a_\sigma & otherwise \end{cases}$

and k is the number of non-leaf elements (excluding the root element)

Note that $'a_i$ is a free type variable, this ensures that we do not accidentally refer to any bound type variable.

As an example, the multiple inheritance sub-typing graph (Figure 4.4b) can be encoded as follows:

```
S = {IC, Num, Cmp, Int, Float, Char, String, Bool}

⟨IC⟩_C = (unit * (unit z * (unit z * 'a)))}
⟨Num⟩_C = (unit z * (unit * (unit z * 'a)))
⟨Cmp⟩_C = (unit z * (unit z * (unit * 'a)))
⟨Int⟩_C = (unit * (unit * (unit * int)))
⟨Float⟩_C = (unit z * (unit * (unit * float)))
⟨Char⟩_C = (unit * (unit z * (unit * char)))
⟨String⟩_C = (unit z * (unit z * (unit * string)))
⟨Bool⟩_C = (unit z * (unit z * (unit z * bool)))

⟨IC⟩_A = (unit * ('a1 * ('a2 * 'a3)))
⟨Num⟩_A = ('a1 * (unit * ('a2 * 'a3)))
⟨Cmp⟩_a = ('a1 * ('a2 * (unit * 'a3)))
⟨Int⟩_A = (unit * (unit * (unit * int)))
⟨Float⟩_A = ('a * (unit * (unit * float)))
⟨Char⟩_A = (unit * ('a * (unit * char)))
⟨String⟩_A = ('a1 * ('a2 * (unit * string)))
⟨Bool⟩_A = ('a1 * ('a2 * ('a3 * bool)))
```

Based on these encodings, module Phantom (Figure 4.5) defines a number of type-safe operations over union types, in contrast with the unsafe ones in Figure 4.3.

An alternative approach to well typed function overloading: Haskell type classes

Phantom types introduce additional free type variables in the data type construction. Such unused phantom parameters provides the freedom of encoding additional information for types. In QQL, this phantom parameter instructs the type system to enforce the difference between subtypes, and provides a typesafe way of handling function overloading in standard OCaml.

Languages such as Haskell offer alternative solutions to this problem – function overloading, especially ad-hoc polymorphism [WB89], which occurs when a function is defined over several different types and acts in a different way for each types. In contrast with standard ML that merely supports simple ad-hoc polymorphism like equality, Haskell adopts a completely new technique, *type classes* [WB89, HHJW94], which extend the familiar Hindley-Milner system to support wide, user-defined ad-hoc polymorphisms.

Type classes in Haskell use an approach similar to object-oriented programming. That is, every type carries with it a pointer to a procedure for performing the overloaded function. If it is allowed to have more than one function with this property, then each type should carry with it a pointer to a directory of appropriate functions. Based on the compile-time translation, application of an overloaded function would be transformed to application of the referenced procedure that is associated with the specific type.

Say that we want to overload arithmetic functions over numerical type Int and Float (c.f. Figure 4.4). We introduce a new *type class*, `Num`. The declaration reads that "*a type* `a` *belongs to class* `Num` *if there are function* `(+)` *and* `(*)` *defined on it*". Type `Int` and `Float` can be declared

```
module Phantom : sig
  type 'a t

  val mkInt : int -> ⟨Int⟩_C t
  val mkFloat : float -> ⟨Float⟩_C t
  val mkChar : char -> ⟨Char⟩_C t
  val mkString : string -> ⟨String⟩_C t
  val mkBool : bool -> ⟨Bool⟩_C t

  val plus : ⟨Num⟩_A t -> ⟨Num⟩_A t -> ⟨Num⟩_A t
  val nextval : ⟨IC⟩_A t -> ⟨IC⟩_A t
end = struct

  type 'a t = Int of int | Float of float | Char of char | Bool of bool | String of string

  (* phantom transformation functions for various types *)
  let mkInt = fun i -> Int i
  let mkFloat = fun f -> Float f
  let mkChar = fun c -> Char c
  let mkString = fun s -> String s
  let mkBool = fun b -> Bool b

  (* Note: the last pattern-matching entry in function plus and nextval will never be
     executed, since the module signature restricts the parameters of plus function to
     two numerical values of the same type, and the parameter of nextval function to
     values of type int or char. These entries exist there only for eliminating OCaml's
     non-exhaustive pattern-matching warning.
  *)
  let plus a b = match a, b with
    |(Int a), (Int b) -> Int (a + b)
    |(Float a), (Float b) -> Float (a +. b)
    |_ -> failwith "unreachable place"

  let nextval = function
    |Int i -> Int (i+1)
    |Char c -> Char (Char.chr ((Char.code c)+1))
    |_ -> failwith "unreachable place"
end
```

Figure 4.5: Type safe operations on union types

as instances of class `Num` by giving appropriate bindings for the functions `(+)` and `(*)`. The type inference algorithm verifies that these bindings obey the type declaration of class `Num`, i.e. `add_int: int -> int -> int, mult_int: int -> int -> int`. (Assume function `add_int`, `mult_int`, `add_float` and `mult_float` are defined in the standard prelude.)

```
class Num a where
  ( + ) :: a -> a -> a
  ( * ) :: a -> a -> a

instance Num Int where
  ( + ) = add_int
  ( * ) = mult_int

instance Num Float where
  ( + ) = add_float
  ( * ) = mult_float
```

This form of overloading is achieved by translating the program into an equivalent program that does not have any overloading. For each class declaration, a dictionary structure (e.g.`NumD`)

and functions (e.g.`add` and `mult`) to access the procedures in the dictionary are generated. Thus, each instance of the class `Num` is translated into the declaration of a value of `NumD`. For instance, a dictionary value `numDInt` containing the bindings for `(+)` and `(*)` is created for `Num Int`, and similarly for `Num Float`. An expression of the form `x + y` is replaced by a corresponding expression `add numD x y` where `numD` is an appropriate dictionary for the type of value `x`. For example, we get the following translation: `1 + 1 => add numDInt 1 1`

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a)

add (NumDict a m) = a
mult (NumDict a m) = m

numDInt :: NumD Int
numDInt = NumDict add_int mult_int

numDFloat :: NumD Float
numDFloat = NumDict add_float mult_float
```

Type classes provide an alternative solution to function overloading — at least in those languages that support it. The feature of class type inheritance and multi-parameter type classes make it a more flexible and easy to use solution to ad-hoc polymorphism than the approach of using phantom types. In addition to function overloading, multi-parameter type classes in combination with functional dependencies can be used for compile-time type calculation, as described by Silva and Visser in [SV06], where they proposed an approach to modelling relational database and operations in Haskell (see Section 2.9).

Type classes, however, cannot be viewed as a replacement for phantom types in all cases, even though it can provide the kind of type-safe function overloading that we require for Qanat. Quite aside from the fact that OCaml, our chosen host language, does not support type classes, there are still situations in which phantom types cannot be replaced by type classes, such as for the extra type constraints required by the readonly/readwrite file access example we have demonstrated at the beginning of this section and the more sophisticated protocol-safe typing constraints we use for transactions, as described in Chapter 5.

4.3.2 Phantom type transformation

In this part, we define the type translation from standard OCaml types to phantom types. The following rules define phantom type translation, PJ K, of arbitrary types in terms of $\langle\tau_C\rangle_C$ and $\langle\tau_A\rangle_A$, where τ_C is a primitive type, e.g. *int*, *float*, *bool*, and τ_A is an abstract type, e.g. *Num*, *Cmp*, *IC*.

$\mathrm{PJ}\tau\mathrm{K}$	$= \langle\tau\rangle_C$, *where* τ *is a primitive type.*
$\mathrm{PJ}\tau\mathrm{K}$	$= \langle\tau\rangle_A$, *where* τ *is an abstract type.*
$\mathrm{PJ}\tau_1 \to \tau_2\mathrm{K}$	$= \mathrm{PJ}\tau_1\mathrm{K} \to \mathrm{PJ}\tau_2\mathrm{K}$
$\mathrm{PJ}\tau$ *option*K	$= \mathrm{PJ}\tau\mathrm{K}$ *option*
$\mathrm{PJ}\tau_1 \times ... \times \tau_n\mathrm{K}$	$= \mathrm{PJ}\tau_1\mathrm{K} \times ... \times \mathrm{PJ}\tau_n\mathrm{K}$
$\mathrm{PJ}\{r_1 : \tau_1; \ldots; r_n : \tau_n\}\mathrm{K}$	$= \{r_1 : \mathrm{PJ}\tau_1\mathrm{K}; \ldots; r_n : \mathrm{PJ}\tau_n\mathrm{K}\}$
$\mathrm{PJ}C_1\langle of\ \tau_1^1, ..., \tau_1^j\rangle \mid \ldots \mid C_n\langle of\ \tau_n^1, \ldots, \tau_n^k\rangle\mathrm{K}$	$= C_1\langle of\ \mathrm{PJ}\tau_1^1\mathrm{K}, \ldots, \mathrm{PJ}\tau_1^j\mathrm{K}\rangle \mid \ldots \mid C_n\langle of\ \mathrm{PJ}\tau_n^1\mathrm{K}, \ldots, \mathrm{PJ}\tau_n^k\mathrm{K}\rangle$
$\mathrm{PJ}<m_1 : \tau_1; \ldots; m_n : \tau_n>\mathrm{K}$	$= <m_1 : \mathrm{PJ}\tau_1\mathrm{K}; \ldots; m_n : \mathrm{PJ}\tau_n\mathrm{K}>$
$\mathrm{PJ}\tau$ *collection*K	$= \mathrm{PJ}\tau\mathrm{K}$ *collection*

In the above conversion rules, the primitive types are transformed into the corresponding phantom types, such as *int* to *Phantom.Types.int*, *string* to *Phantom.Types.string*, etc. The conversion of composition types, such as tuples, records, etc., is performed by transforming each of its encapsulated elements.

4.3.3 Shadowing QQL functions with phantom functions

Phantom types uses additional type variables that are free (unused) in the type definition to track additional information. This makes it possible to distinguish concrete subtypes, achieving type-safe function overloading in an ML-like type system. By employing phantom types, we can define aggregate functions, such as `sum` and `avg` on numerical values (e.g. int and float), `min` and `max` on comparable values (e.g. int, string), in a way similar to function `plus` defined in Figure 4.5.

We provide shadow functions for each built-in function that can appear in a query, which we will call *query functions*, for the purpose of type calculation within a phantom context. These include not just the usual aggregate functions, but also functions for clauses in the query language, such as `select`, `from` and `where`. Based on the phantom encoding in section 4.3.1, primitive types can be encoded as `('a * ('b * 'c)) Phantom.t`, where type variable `'a` represents whether this type is numeric or not, `'b` indicates whether it is comparable, and `'c` is the type before the transformation. To improve readability and error reporting, we use the polymorphic variants `` `Numerical `` and `` `Unknown `` to represent the boolean value for the numeric case, instead of `unit` and `unit z`. A similar representation is used for encoding the comparable information. Taking the aggregate function as example, function `sum` iterates over all values present in a numeric collection, calculating the sum of these values and returning a result that must be of the same type with the collection element. The type of `sum` is therefore captured as:

```
val sum: (([`Numerical] * ('b * 'c)) Phantom.t) collection ->
                                  ([`Numerical] * ('b * 'c)) Phantom.t
```

Function `max` returns the maximum value from a collection, the elements of which must be comparable,

```
val max: (('a * ([`Comparable] * 'c)) Phantom.t) collection ->
                                  ('a * ([`Comparable] * 'c)) Phantom.t
```

while function `count` can be performed on any collection, returning an integer value,

```
val count: 'a collection -> (`[Numerical] * ([`Comparable] * int)) Phantom.t
```

Recall that these query functions will only be executed by the database server, not by the host programming language, and we are only defining types for them so that we can do type checking across OCaml code and QQL database queries that invoke the query functions. Hence, we define these types by providing dummy implementations of query function that have the appropriate type information, whereas the actual values returned by these dummy functions are immaterial, as they will never be invoked. We will show later how we use these dummy functions to do the type checking of QQL queries.

```
let count (xs: 'a collection) = Phantom.mkInt 0
let sum (xs: (([`Numerical] * 'b * 'c) Phantom.t) collection) = Collection.hd xs
let avg (xs: (([`Numerical] * 'b * 'c) Phantom.t) collection) = Collection.hd xs
let min (xs: (('a * ([`Comparable] * 'c)) Phantom.t) collection) = Collection.hd xs
let max (xs: (('a * ([`Comparable] * 'c)) Phantom.t) collection) = Collection.hd xs
```

QQL supports two kinds of aggregation expressions:

- Standard Aggregation, based on `group-by`, resembles SQL's aggregation. That is, aggregate functions are performed over sets of values partitioned by a `group-by` clause. Where `fn` is an aggregation function, it is expressed as:

  `select k, fn` x_p`#e from` $x_1, \ldots, x_n$ `where E group by` x_i`#v as k`

- Field-Expression Aggregation, based on *field-access-expression* views, as a natural form for expressing aggregation within an object-oriented context. That is, aggregate functions are invoked on collection data as an object method, expressed as;

  `select e#fn () from` $x_1, \ldots, x_n$ `where E`

  where `e` represents a collection value that can be obtained directly from one object (e.g. `cust#orders`), or derived from the mapping action (e.g. `cust#orders#[x => x#charge]`).

For example, consider this code running against a bookshop database:

```
select c#email, c#orders#[x => x#charge]#max() from customer as c
```

This query finds the maximum charge of orders placed by each customer. Expression `c#orders#[x => x#charge]` picks out the charge of each order placed by the customer, equivalent to `map (fun x -> x#charge) c#orders`; then the aggregate function `max` is performed on these charges. This is different from group-by aggregation in that the field-expression aggregation ends up as a sub-query. For instance, the above query will be translated to the following SQL:

```
select t1.email, (select max(t3.charge)
                  from customers t2 left join orders t3 on t2.id = t3.custid
                  where t2.id = t1.id)
from customers t1
```

Until now, we have only discussed QQL aggregation over numerical and comparable values. However, SQL aggregation can be applied to nullable values and return a NULL as result. To handle such cases, it is necessary to extend QQL aggregation to optional values as well. For example, function `max` is performed over a collection of comparable values. If optional type τ *option* can track the comparable information of τ, the same `max` can be applied to a collection of optional data. To achieve this, function `optionize` for converting optional values into phantom data is defined as,

```
let optionize : (('num *('cmp * 't))) Phantom.t) option ->
      ('num * ('cmp * (('num * ('cmp * 't)) Phantom.t) option)) Phantom.t =
      fun x -> Obj.repr x
```

where `optionize` transforms an optional value to a phantom value, while inheriting the same numerical and comparable information; `get` performs the inverse action. Consider a phantom integer with the encoding $int_{phantom}$,

```
type int_phantom = ([`Numerical] * ([`Comparable] * int)) Phantom.t
```

the application of `optionize` on $int_{phantom}$ *option* results in

```
([`Numerical] * ([`Comparable] * (int_phantom option))) Phantom.t
```

which inherits enough information for aggregate functions. By converting optional data into phantom data that keeps the same numerical and comparable information, aggregate functions in QQL can be applied to collections of both primitive-type and option-type values.

The implementation of the phantom module indicated in Figure 4.5 is unnecessarily restrictive. For our purposes, we do not need real implementations of the aggregate methods: we need only capture their type constraints. Thus the use of union types for the base type forces us to list the individual types, and modify the code to add new ones as they are required. Similarly, we would need to explicitly implement the code for each phantom type to flatten it back to the base type. Instead we take a simpler approach by using the values' internal representations, rather than using union types. This change helps achieve a concise phantom transformation, namely generic functions for converting between standard value and phantom-typed values. Specifically, the implementation uses OCaml's `Obj` module, which defines polymorphic functions (e.g. `repr` and `obj`) for converting values into and back from the internal representation respectively, corresponding to `init` and `get` that we define in Figure 4.6. Function `init` initialises a value into a phantom-typed value; `get` extracts the value from phantom data; `numerical` declares a phantom value as numerical by setting tag *numerical*; `comparable` is defined as an analog to `numerical`. The phantom transformation function for a certain type can be defined by composing these functions, i.e. `init`, `numerical` and `comparable`.

The construction of shadow query functions for type calculation, including basic select-from-where, arithmetic, comparison and logic operations, etc., normally requires the design of a number of polymorphic but constrained functions. For instance, `from` defines query sources that can be different ORM classes, which we will call `queryable`; `group-by` and `order-by` can only be applied on non-collection type fields, which we will call *univalued*. To capture these constraints, the phantom encoding of QQL needs to be extended by introducing two categories, *queryable* and *univalued*: queryable indicates that this value is an ORM class that can be referred to in a `from`

```
module Phantom : sig
  type 'a t

  val mkInt : int -> ([`Numerical] * ([`Comparable] * int)) t
  val mkFloat : float -> ([`Numerical] * ([`Comparable] * float)) t
  val mkChar : char -> ([`Unknown] * ([`Comparable] * char)) t
  val mkString : string -> ([`Unknown] * ([`Comparable] * string)) t
  val mkBool : bool -> ([`Unknown] * ([`Unknown] * bool)) t

end = struct

  type 'a t = Obj.t

  (* initialize a phantom type *)
  let init : 'a -> ([`Unknown] * ([`Unknown] * 'a)) t = Obj.repr
  (* inverse operation to init *)
  let get : ('a * ('b * 'c)) t -> 'c = Obj.obj

  (* set numerical tag for a phantom-typed value *)
  let numerical : ('a * ('b * 'c)) t -> ([`Numerical] * ('b * 'c)) t = fun x -> x
  (* set comparable tag for a phantom-typed value *)
  let comparable : ('a * ('b * 'c)) t -> ('a * ([`Comparable] * 'c)) t = fun x -> x

  (* phantom transformation functions for various types *)
  let mkInt = fun i -> numerical (comparable (init i))
  let mkFloat = fun f -> numerical (comparable (init f))
  let mkChar = fun c -> comparable (init c)
  let mkString = fun s -> comparable (init s)
  let mkBool = fun b -> init b
end
```

Figure 4.6: Phantom module implementation in OCaml

clause, while univalued indicates that this value is not of any collection type. The resulting general phantom type is, therefore:

```
('numerical * ('comparable * ('queryable * ('univalued * 'type')))) Phantom.t
```

Using this encoding, composition type values, other than *collection*, are encapsulated into phantom-typed values. Before we can present the new, and final, version of the phantom type transformation rules, we first need to introduce some phantom type manipulation functions.

- Phan (n, c, q, u, τ) constructs a phantom type where the contained type is τ, n can be `'numerical` or `'unknown`, indicating whether or not the phantom type is a subtype of *numerical*, with similar meanings for c and *comparable*, q and *queryable*, and u and *univalued*. To help fit the phantom type transformation on the page, we rename, just for this figure, `'numerical`, `'comparable`, `'queryable`, `'univalued` and `'unknown` to `'num`, `'cmp`, `'qry`, `'uni`, and `'not` respectively.

- Num(τ) returns the *numerical* status of phantom type τ, i.e. it returns `'num`, if τ is a subtype of *numerical*, and `'not` otherwise.

- Cmp(τ) returns the *comparable* status of phantom type τ, i.e. it returns `'cmp`, if τ is a subtype of *comparable*, and `'not` otherwise.

The modified phantom type transformation rules are presented in Figure 4.7. The significance of the modification is that all QQL composition types are now encapsulated as phantom types. The rule for the *object* type is split into the ones for anonymous objects and ORM classes. ORM classes are subtypes of the `queryable` abstract phantom type, indicating that this class can be used

$$
\begin{array}{ll}
\mathrm{P\,J}\tau\mathrm{K} & = \langle\tau\rangle_C, \textit{ where } \tau \textit{ is a primitive type.} \\
\mathrm{P\,J}\tau\mathrm{K} & = \langle\tau\rangle_A, \textit{ where } \tau \textit{ is an abstract type.} \\
\mathrm{P\,J}\tau_1 \to \tau_2\mathrm{K} & = \mathrm{P\,J}\tau_1\mathrm{K} \to \mathrm{P\,J}\tau_2\mathrm{K} \\
\mathrm{P\,J}\tau \textit{ option}\mathrm{K} & = \mathrm{Phan}\,(\mathrm{Num}(\mathrm{P\,J}\tau\mathrm{K}), \mathrm{Cmp}(\mathrm{P\,J}\tau\mathrm{K}), \texttt{'not}, \texttt{'uni}, \mathrm{P\,J}\tau\mathrm{K}\ \textit{option}) \\
\mathrm{P\,J}\tau_1 \times ... \times \tau_n\mathrm{K} & = \mathrm{Phan}\,(\texttt{'not}, \texttt{'not}, \texttt{'not}, \texttt{'uni}, \mathrm{P\,J}\tau_1\mathrm{K} \times ... \times \mathrm{P\,J}\tau_n\mathrm{K}) \\
\mathrm{P\,J}\{r_1 : \tau_1; \ldots; r_n : \tau_n\}\mathrm{K} & = \mathrm{Phan}\,(\texttt{'not}, \texttt{'not}, \texttt{'not}, \texttt{'uni}, \{r_1 : \mathrm{P\,J}\tau_1\mathrm{K}\,; \ldots; r_n : \mathrm{P\,J}\tau_n\mathrm{K}\}) \\
\mathrm{P\,J}\begin{array}{l} C_1\langle \textit{of } \tau_1^1, \ldots, \tau_1^j\rangle | \ldots \\ \quad \ldots | C_n\langle \textit{of } \tau_n^1, \ldots, \tau_n^k\rangle \end{array}\mathrm{K} & = \mathrm{Phan}\left(\texttt{'not}, \texttt{'not}, \texttt{'not}, \texttt{'uni}, \begin{array}{r} C_1\langle \textit{of } \mathrm{P\,J}\tau_1^1\mathrm{K}, \ldots, \mathrm{P\,J}\tau_1^j\mathrm{K}\rangle | \ldots \\ \quad \ldots | C_n\langle \textit{of } \mathrm{P\,J}\tau_n^1\mathrm{K}, \ldots, \mathrm{P\,J}\tau_n^k\mathrm{K}\rangle \end{array}\right) \\
\mathrm{P\,J}{<}m_1 : \tau_1; \ldots; m_n : \tau_n{>}\mathrm{K} & = \mathrm{Phan}\,(\texttt{'not}, \texttt{'not}, \texttt{'not}, \texttt{'uni}, {<}m_1 : \mathrm{P\,J}\tau_1\mathrm{K}\,; \ldots; m_n : \mathrm{P\,J}\tau_n\mathrm{K}{>}) \\
\mathrm{P\,J}\texttt{orm}\ {<}m_1 : \tau_1; \ldots; m_n : \tau_n{>}\mathrm{K} & = \mathrm{Phan}\,(\texttt{'not}, \texttt{'not}, \texttt{'qry}, \texttt{'uni}, \texttt{orm}\ {<}m_1 : \mathrm{P\,J}\tau_1\mathrm{K}\,; \ldots; m_n : \mathrm{P\,J}\tau_n\mathrm{K}{>}) \\
\mathrm{P\,J}\tau \textit{ collection}\mathrm{K} & = \mathrm{P\,J}\tau\mathrm{K}\ \textit{collection} \\
\mathrm{P\,J}\tau \textit{ query}\mathrm{K} & = \mathrm{P\,J}\tau\mathrm{K}\ \textit{query}
\end{array}
$$

Figure 4.7: Refreshed Phantom Type Transformation Rules

as a query source. Thus the QQL type structure, including the full QQL abstract type hierarchy, is mirrored in the phantom type structure.

Lemma 4.2 *Let $\mathcal{T}_{phantom}$ be the phantom QQL type system, $\mathcal{T}$ be the standard QQL type system. If there is a type function σ such that, for every primitive or abstract type t in $\mathcal{T}$, $\sigma\,(\mathrm{P\,J}t\mathrm{K}) = t$, then there is an inverse type transformation $\mathrm{P}^{-1}\mathrm{J}\ \mathrm{K}$ such that, for every type t' in $\mathcal{T}$, $\mathrm{P}^{-1}\mathrm{J}\ \mathrm{P\,J}t'\mathrm{K}\ \mathrm{K} = t$.*

Proof: By construction: use σ for transformed primitive types. The inverse transform for composite types can be performed by recursively transforming each of its encapsulated elements.

Lemma 4.3 (Inversion of Phantom QQL Types Transformation) $\mathrm{P\,J}\cdot\mathrm{K}$ *is invertible.*

Proof: The mapping from QQL types to phantom types is bijective for primitive types, which can be seen by an inspection of the type encoding. Therefore the necessary inverse type transform σ' required by Lemma 4.2 exists and we can use that lemma to obtain the required result.

```
let queryable :
  ('a*('b*('c*('d*'t')))) Phantom.t -> ('a*('b*([`Queryable]*('d*'t')))) Phantom.t =
  fun x -> x

let univalued :
  ('a*('b*('c*('d*'t')))) Phantom.t -> ('a*('b*('c*([`Univalued]*'t')))) Phantom.t =
  fun x -> x

let optionize : (('a*('b*'x'))) Phantom.t) option ->
  ('a*('b*([`Unknown]*([`Unknown]*( (('a*('b*'x')) Phantom.t) option ))))) Phantom.t =
  fun x -> Obj.repr x

Phantom.get (init x) = x
Phantom.get (optionize x) = x
Phantom.get (univalued (init x)) = x
Phantom.get (queryable (init x)) = x
```

Figure 4.8: Utility functions for the Phantom Value Transformation function

```
pvt(int) = fun v -> Phantom.mkInt v
pvt(float) = fun v -> Phantom.mkFloat v
...
pvt(τ option) = fun v -> Phantom.univalued(
                          Phantom.optionize(match v with
                                              |Some x -> Some (pvt(τ) x)
                                              |None -> None))
pvt(τ1 × ··· × τn) = fun v -> let (x1, ···, xn) = v in
        Phantom.univalued(Phantom.init (pvt(τ1) x1, ···, pvt(τn) xn))
pvt({r1 : τ1; ··· ; rn : τn}) = fun v ->
        Phantom.univalued(Phantom.init {r1 = pvt(τ1) v.r1; ···; rn = pvt(τn) v.rn})
pvt(C1⟨of τ1^1, ..., τ1^j⟩| ..... |Cn⟨of τn^1, ..., τn^k⟩) = fun v ->
       Phantom.univalued(Phantom.init (match v with
                    |C1⟨of v1^1, ..., v1^j⟩ -> C1⟨of pvt(τ1^1) v1^1, ..., pvt(τ1^j) v1^j⟩
                    |...
                    |Cn⟨of vn^1, ..., vn^k⟩ -> Cn⟨of pvt(τn^1) vn^1, ..., pvt(τn^k) vn^k⟩)))
pvt(<m1 : τ1; ...; mn : τn>) = fun v ->
        Phantom.univalued(Phantom.init(
            object method m1 = pvt(τ1) v#m1; ···; method m1 = pvt(τn) v#mn end))
pvt(orm <m1 : τ1; ...; mn : τn>) = fun v ->
        Phantom.queryable(Phantom.univalued(Phantom.init(
            object method m1 = pvt(τ1) v#m1; ···; method m1 = pvt(τn) v#mn end)))
pvt(τ collection) = fun v -> map (fun x -> pvt(τ) x) v
pvt(τ query) = fun v -> {v with resultset = pvt(τ) v.resultset}
```

Figure 4.9: The Phantom Value Transformation function

4.4 The QQL phantom type system

At this point we can now present the QQL phantom type system.

phantomtype = *t* | *(t collection) query*

t = P J*t*K | *orm-type* | *(t) collection*

basic-type =
- *primitive-type*
- *primitive-type option* (* *option type* *)
- $basic\text{-}type_1 \times \cdots \times basic\text{-}type_n$ (* *tuple type* *)
- $\{r_1 : basic\text{-}type_1; \cdots ; r_n : basic\text{-}type_n\}$ (* *record type* *)
- $\langle m_1 : basic\text{-}type_1; \cdots ; m_n : basic\text{-}type_n\rangle$ (* *anonymous object type* *)
- $\left[C_1 \texttt{ of } basic\text{-}type_1^1 \times \cdots \times basic\text{-}type_1^j | \cdots | C_n \texttt{ of } basic\text{-}type_n^1 \times \cdots \times basic\text{-}type_n^k\right]$ (* *variant type* *)

primitive-type = *int, float, char, bool, ...*

orm-type = P J*orm* $\langle m_1 : t_1; \cdots ; m_n : t_n\rangle$K | P J*(orm* $\langle m_1 : t_1; \cdots ; m_n : t_n\rangle$*) option*K

We will use the same names for the abstract types in the QQL phantom type system as for the QQL type system and rely on context to distinguish between them.

Numerical = *([‘Numerical] * (’b * (’c * (’d * ’x))) Phantom.t*

Comparable = *(’a * ([‘Comparable] * (’c * (’d * ’x))) Phantom.t*

Queryable = *(’a * (’b * ([‘Queryable] * (’d * ’x))) Phantom.t*

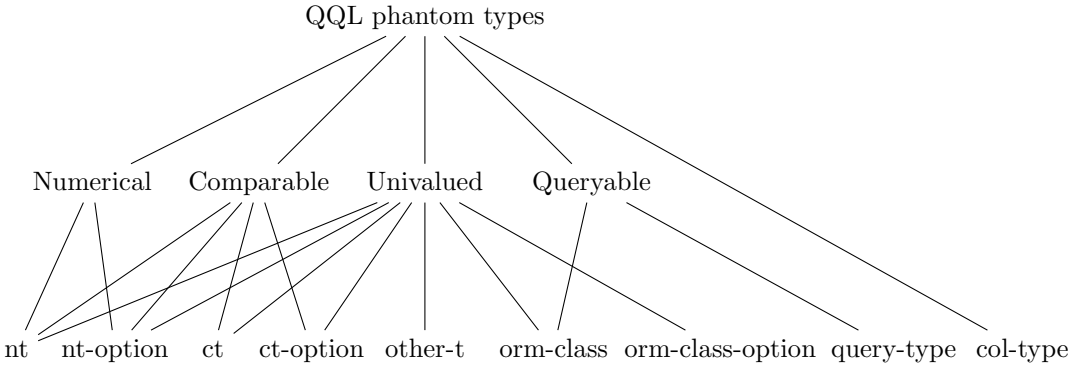
| *(t collection) query*

Univalued = *('a * ('b * ('c * ([‘Univalued] * 'x))) Phantom.t*

QQLTypes = *Numerical* | *Comparable* | *Queryable* | *Univalued* | *(t) collection*

The QQL phantom type system mirrors the normal QQL type system. Types in the phantom type system is transformed from standard QQL types by using the phantom transformation functions defined in section 4.3.3. The type transformation function P J·K was defined in Figure 4.7. To capture the sub-typing graph *Numerical*, *Comparable*, *Univalued* and *Queryable* of types, the phantom encoding in QQL includes four additional type variables, in addition to the last type variable storing the original type information. For all numerical values, the *Numerical* type variable of the phantom encoding is set, thus these values can be used in functions that takes parameters of the type `([‘Numerical] * ('b * ('c * ('d * 'x))) Phantom.t`. Similarly, type variables in the phantom encoding for *Comparable*, *Queryable* and *Univalued* values are set. *Queryable* types include all orm classes and user-defined queries: for orm classes, the corresponding type variable in the phantom encoding is set as *true*, i.e. `[‘Queryable]`; while queries are treated in a different way in the typing rules. This is possible since query-type variables in the `from` clause are denoted differently from orm classes, thus different typing rules can be applied.), c.f. rule pSFW.

We present a diagram of QQL phantom types in Figure 4.10, which can be viewed as a "phantomised" diagram of Figure 4.2.



`nt`	$= \mathrm{P\,J}int\mathrm{K} \mid \mathrm{P\,J}float\mathrm{K} \mid \mathrm{P\,J}int32\mathrm{K} \mid \cdots \mid \mathrm{P\,J}Numeric.t\mathrm{K}$
nt-option	$= \mathrm{P\,J}int\ option\mathrm{K} \mid \mathrm{P\,J}float\ option\mathrm{K} \mid \mathrm{P\,J}int32\ option\mathrm{K} \mid \cdots \mid \mathrm{P\,J}Numeric.t\ option\mathrm{K}$
`ct`	$= \mathrm{P\,J}char\mathrm{K} \mid \mathrm{P\,J}string\mathrm{K} \mid \cdots \mid \mathrm{P\,J}timetz\mathrm{K}$ (time with time-zone)
ct-option	$= \mathrm{P\,J}char\ option\mathrm{K} \mid \mathrm{P\,J}string\ option\mathrm{K} \mid \cdots \mid \mathrm{P\,J}timetz\ option\mathrm{K}$ (time with time-zone)
other-t	$= \mathrm{P\,J}bool\mathrm{K} \mid \mathrm{P\,J}bool\ option\mathrm{K} \mid \mathrm{P\,J}$std-record-type$\mathrm{K} \mid \mathrm{P\,J}$std-tuple-type$\mathrm{K}$ $\mid \mathrm{P\,J}$std-variant-type$\mathrm{K} \mid \mathrm{P\,J}$std-anonymous-object$\mathrm{K}$
orm-class	$= \mathrm{P\,J}orm < m_1 : t_1; \cdots ; m_n : t_n >\mathrm{K}$
orm-class-option	$= \mathrm{P\,J}(orm < m_1 : t_1; \cdots ; m_n : t_n >)\ option\mathrm{K}$
query-type	$= (t\ collection)\ query$
col-type	$= (t)\ collection$

Figure 4.10: Phantom Types

4.4.1 QQL phantom typing rule

We first present typing rules for QQL expressions under the phantom type system, then give the typing rules for select queries. The following typing rules are quite close to the ones in section 4.2.2, except that type constraints are captured here by using phantom encoding. Hence we omit explanations that were given in that section and which apply, in an obvious way, to this one.

$$\frac{}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{i} : \mathrm{P}\,\mathrm{J}\mathit{int}\mathrm{K}}\ \textsc{p}\textsc{Lit-int} \qquad \frac{}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{s} : \mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K}}\ \textsc{p}\textsc{Lit-string} \qquad \ldots$$

$$\frac{}{\mathrm{S};\mathrm{D},\mathrm{v}:\tau;\Gamma \vdash_p \texttt{:v} : \tau}\ \textsc{p}\textsc{Var}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{x} : \mathrm{P}\,\mathrm{J}< m_1 : t_1; \cdots ; m_n : t_n >\mathrm{K} \qquad 1 \le i \le n}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{x\#}m_i : \mathrm{P}\,\mathrm{J}t_i\mathrm{K}}\ \textsc{p}\textsc{Object field access}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{x} : \mathrm{P}\,\mathrm{J}\{r_1 : t_1; \cdots ; r_n : t_n\}\mathrm{K} \qquad 1 \le i \le n}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{x.}r_i : \mathrm{P}\,\mathrm{J}t_i\mathrm{K}}\ \textsc{p}\textsc{Record field access}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{vs} : \tau\ \mathit{collection} \qquad \mathrm{S};\mathrm{D};\Gamma, \mathrm{x} : \tau \vdash_p \texttt{e} : \sigma}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{vs\#[x => e]} : \sigma\ \mathit{collection}}\ \textsc{p}\textsc{Collection field mapping}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{e} : \mathit{('a\ *\ ('b\ *\ ('c\ *\ ('d\ *\ ((('a\ *\ ('b\ *\ 't))Phantom.t)\ option)))))\ Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{valof e} : \mathit{('a\ *\ ('b\ *\ 't))\ Phantom.t}}\ \textsc{p}\textsc{Valof}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{e} : \tau\ \mathit{collection}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{itemsof e} : \tau}\ \textsc{p}\textsc{Itemsof}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p x_1 : \mathrm{P}\,\mathrm{J}t_1\mathrm{K} \qquad \cdots \qquad \mathrm{S};\mathrm{D};\Gamma \vdash_p x_n : \mathrm{P}\,\mathrm{J}t_n\mathrm{K}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p (x_1, \cdots, x_n) : \mathrm{P}\,\mathrm{J}t_i \times \cdots \times t_n\mathrm{K}}\ \textsc{p}\textsc{tuple}$$

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{arith}_{\mathit{OP}} : \tau \to \tau \to \tau \\ \quad \tau <: \mathit{(['Numerical]\ *\ ('b\ *\ ('c\ *\ ('d\ *\ 't))))\ Phantom.t}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1\ \texttt{arith}_{\mathit{OP}}\ e_2 : \tau}\ \textsc{p}\textsc{Arithmetic}_{\texttt{op}}$$

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \tau \qquad \tau <: \mathit{('a\ *\ (['Comparable]\ *\ ('c\ *\ ('d\ *\ 't))))\ Phantom.t}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1\ \texttt{compare}_{\mathit{OP}}\ e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}\ \textsc{p}\textsc{Comparison}_{\texttt{op}}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K} \qquad \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1\ \texttt{and}\ e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}\ \textsc{p}\textsc{And}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K} \qquad \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1\ \texttt{or}\ e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}\ \textsc{p}\textsc{Or}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{e} : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{not e} : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}\ \textsc{p}\textsc{Not}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau \qquad \tau <: \textit{(['Numerical] * ('b * ('c * ('d * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{sum\ e} : \tau}\ \textsc{pSum}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau \qquad \tau <: \textit{(['Numerical] * ('b * ('c * ('d * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{avg\ e} : \tau}\ \textsc{pAvg}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau \qquad \tau <: \textit{('a * (['Comparable] * ('c * ('d * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{max\ e} : \tau}\ \textsc{pMax}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau \qquad \tau <: \textit{('a * (['Comparable] * ('c * ('d * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{min\ e} : \tau}\ \textsc{pMin}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \alpha}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{count\ e} : \mathrm{P}\,[\![\mathit{int}]\!]}\ \textsc{pCount} \qquad \frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \alpha\ \textit{collection}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e\#count()} : \mathrm{P}\,[\![\mathit{int}]\!]}\ \textsc{pCount-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau\ \textit{collection} \qquad \tau <: \textit{(['Numerical] * ('b * ('c * ('d * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e\#sum()} : \tau}\ \textsc{pSum-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau\ \textit{collection} \qquad \tau <: \textit{(['Numerical] * ('b * ('c * ('d * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e\#avg()} : \tau}\ \textsc{pAvg-col}$$

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau\ \textit{collection} \\ \quad \tau <: \textit{('a * (['Comparable] * ('c * ('d * 't)))) Phantom.t}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e\#max()} : \tau}\ \textsc{pMax-col}$$

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e} : \tau\ \textit{collection} \\ \quad \tau <: \textit{('a * (['Comparable] * ('c * ('d * 't)))) Phantom.t}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \mathtt{e\#min()} : \tau}\ \textsc{pMin-col}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \tau \qquad \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \tau \qquad \cdots}{\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{[}e_1\texttt{; } e_2\texttt{; } \cdots\texttt{]} : \tau \textit{ collection}}\ \textsc{pList}$$

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \tau \textit{ collection} \qquad \tau <: \textit{('a * ('b * ('c * ([`Univalued] * 't)))) Phantom.t}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 \texttt{ in } e_2 : \mathrm{P\,J}\textit{bool}\mathrm{K}}\ \textsc{pIn}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \mathrm{P\,J}\textit{string}\mathrm{K} \qquad \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \mathrm{P\,J}\textit{string}\mathrm{K}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 \texttt{ like } e_2 : \mathrm{P\,J}\textit{bool}\mathrm{K}}\ \textsc{pLike}$$

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p e_2 : \tau \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p e_3 : \tau \\ \quad \tau <: \textit{('a * ('b * ('c * ([`Univalued] * 't)))) Phantom.t}\end{array}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e_1 \texttt{ between } e_2 \texttt{ and } e_3 : \mathrm{P\,J}\textit{bool}\mathrm{K}}\ \textsc{pBetween-and}$$

$$\frac{\mathrm{S};\mathrm{D};\Gamma \vdash_p e : \tau \textit{ option} \qquad \tau <: \textit{('a * ('b * ('c * ([`Univalued] * 't)))) Phantom.t}}{\mathrm{S};\mathrm{D};\Gamma \vdash_p e \texttt{ is [not]}^{0.1} \texttt{ Null} : \mathrm{P\,J}\textit{bool}\mathrm{K}}\ \textsc{pIs-null}$$

Rules pSFW and pSFWGH type QQL queries. It is worth noting the differences between the judgements of query-type variables $:p_i$ and orm classes q_i in the `from` clause. The judgement of expression : p_i indicates the identifier v_i of a *Queryable* type ε'_i, which is constructed from ε_i, and added into Γ. While the judgement of expression q_i binds identifier x_i to *Queryable* type σ_i directly. Whereas both typing rules bind a *Queryable* identifier to Γ, these identifiers, along with others in Γ, can be used to construct the query clauses and expressions.

$$\frac{\begin{array}{l}\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{:p}_1 : \textit{(ε_1 collection) query} \\ \quad \varepsilon_1 <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\ \mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_1 : \varepsilon_1 \vdash_p \texttt{:p}_2 : \textit{(ε_2 collection) query} \\ \quad \vdots \\ \mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_{k-1} : \varepsilon_{k-1} \vdash_p \texttt{:p}_k : \textit{(ε_k collection) query} \\ \quad \varepsilon_k <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\ \mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon} \vdash_p \texttt{q}_1 : \sigma_1 \textit{ collection} \\ \quad \sigma_1 <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\ \mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1 \vdash_p \texttt{q}_2 : \sigma_2 \textit{ collection} \\ \quad \sigma_2 <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\ \quad \vdots \\ \mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1, \ldots, \mathrm{x}_{n-1} : \sigma_{n-1} \vdash_p \texttt{q}_n : \sigma_n \textit{ collection} \\ \quad \sigma_n <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\ \mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash_p \texttt{w} : \mathrm{P\,J}\textit{bool}\mathrm{K} \\ \mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash_p \texttt{s} : \tau\end{array}}{\begin{array}{lll} & \texttt{select s from :p}_1 \texttt{ as v}_1 \texttt{ ... , :p}_k \texttt{ as v}_k & (\varepsilon_1 \textit{ collection}) \textit{ query} \to \cdots \to \\ \mathrm{S};\mathrm{D};\Gamma \vdash_p & \texttt{q}_1 \texttt{ as x}_1\texttt{, ... , q}_n \texttt{ as x}_n\texttt{,} & : \ (\varepsilon_k \textit{ collection}) \textit{ query} \to \\ & \texttt{where w} & (\tau \textit{ collection}) \textit{ query}\end{array}}\ \textsc{pSFW}$$

$$\frac{\begin{array}{l}
\mathrm{S};\mathrm{D};\Gamma \vdash_p \texttt{:p}_1 : (\varepsilon_1\ \textit{collection})\ \textit{query} \\
\quad \varepsilon_1 <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\
\mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_1 : \varepsilon_1 \vdash_p \texttt{:p}_2 : (\varepsilon_2\ \textit{collection})\ \textit{query} \\
\quad \vdots \\
\mathrm{S};\mathrm{D};\Gamma, \mathrm{v}_{\mathrm{k}-1} : \varepsilon_{\mathrm{k}-1} \vdash_p \texttt{:p}_k : (\varepsilon_k\ \textit{collection})\ \textit{query} \\
\quad \varepsilon_k <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon} \vdash_p \texttt{q}_1 : \sigma_1\ \textit{collection} \\
\quad \sigma_1 <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1 \vdash_p \texttt{q}_2 : \sigma_2\ \textit{collection} \\
\quad \sigma_2 <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\
\quad \vdots \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \mathrm{x}_1 : \sigma_1, \ldots, \mathrm{x}_{\mathrm{n}-1} : \sigma_{\mathrm{n}-1} \vdash_p \texttt{q}_n : \sigma_n\ \textit{collection} \\
\quad \sigma_n <: \textit{('a * ('b * ([`Queryable] * 'x))) Phantom.t} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash_p \texttt{w} : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma} \vdash_p \texttt{g}_1 : \tau_1 \\
\quad \tau_1 <: \textit{('a * ('b * ('c * ([`Univalued] * 't)))) Phantom.t} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \mathrm{k}_1 : \tau_1 \vdash_p \texttt{g}_2 : \tau_2 \\
\quad \tau_2 <: \textit{('a * ('b * ('c * ([`Univalued] * 't)))) Phantom.t} \\
\quad \vdots \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \mathrm{k}_1 : \tau_1, \ldots, \mathrm{k}_{\mathrm{j}-1} : \tau_{\mathrm{j}-1} \vdash_p \texttt{g}_j : \tau_j \\
\quad \tau_j <: \textit{('a * ('b * ('c * ([`Univalued] * 't)))) Phantom.t} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \bar{\mathrm{k}} : \bar{\tau} \vdash_p \texttt{h} : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K} \\
\mathrm{S};\mathrm{D};\Gamma, \bar{\mathrm{v}} : \bar{\varepsilon}, \bar{\mathrm{x}} : \bar{\sigma}, \bar{\mathrm{k}} : \bar{\tau} \vdash_p \texttt{s} : \varphi
\end{array}}{\begin{array}{lll}
 & \texttt{select s from :p}_1\ \texttt{as v}_1\ \ldots\ \texttt{, :p}_k\ \texttt{as v}_k & \\
 & \texttt{q}_1\ \texttt{as x}_1,\ \ldots\ ,\ \texttt{q}_n\ \texttt{as x}_n, & (\varepsilon_1\ \textit{collection})\ \textit{query} \to \cdots \to \\
\mathrm{S};\mathrm{D};\Gamma \vdash_p & \texttt{where w} & : (\varepsilon_k\ \textit{collection})\ \textit{query} \to \\
 & \texttt{group by g}_1\ \texttt{as k}_1, \ldots, \texttt{g}_j\ \texttt{as k}_j & (\varphi\ \textit{collection})\ \textit{query} \\
 & \texttt{having h} &
\end{array}}\ \textsc{pSFWGH}$$

4.5 Type avatars for QQL type safety

We extend the concept of *shadow functions*, which are predefined for each built-in function in QQL, to the more general *type avatars*, which are shadow data structures capturing the type constraints on entire queries. Whereas shadow functions are fixed by the query language definition, type avatars are generated by the preprocessor at compile time for every QQL query and composable query part. A type avatar is designed as dummy data, generated for each QQL query, to enable compile-time type checking to be carried out by the host language compiler. The reference to 'dummy' data means that avatar data is never evaluated for query execution or other purposes at any time. Essentially, type avatars are generated only to model type constraints existing in the query.

We first present a phantom *value* transformation function $pvt(\cdot)$, c.f. Figure 4.9, that takes a value of a QQL type and returns the corresponding values of phantom type. Based on this function, an orm module (i.e. a session module) is generated with pre-defined phantom transformation functions for managed orm classes. The function $pvt(\cdot)$ uses functions `queryable` and `univalued` to set tag 'queryable' and 'univalued', and `optionize` to inherit numerical and comparable tags, c.f. Figure 4.8.

Note that function `Phantom.get` acts as the inverse to each of `init`, `optionize`, `queryable` and `univalued`, converting phantom values back into the state before the application of these functions. In fact, standard primitive QQL types are transformed into phantom types by putting them at the leaf of a right deep tree of the same structure for all types. Hence we can easily invert that transform, in the case of primitive types, by forgetting all except the leaf of the tree.

Figure 4.12 shows the QQL query function shadows that will be used in the type avatar for type checking, the main ones being:

1. `from` defines query sources and the constraint that the type of these sources is queryable. `join` represents the join of two query sources and takes additional functions for obtaining univalued values that are used for the equality condition from query sources.
2. `where` and `having` define boolean predicates based on the query context that is constructed from query sources.
3. `limit` and `offset` take an integer value for customising the query results.
4. `orderby` specifies an ordering criteria that is based on a comparable field obtained from the query context.
5. `groupby` defines a partition criteria based on a univalued field obtained from the query context. It differs from the other functions in that the `group-by` clause changes the query context. Thus `groupby` returns a univalued field that is used to construct a new context.
6. `select` defines the return value of the query, based on the given context that is either constructed from the `from` clause, or modified by `group-by`.

```
(* Phantom type module *)
module Phantom : sig
  type 'a t

  module Types : sig
    type phantom_int = ([`Numerical]*([`Comparable]*([`Unknown]*([`Univalued]* int)))) t
    type phantom_string = ([`Unknown]*([`Comparable]*([`Unknown]*([`Univalued]* string)))) t
    type phantom_bool = ([`Unknown]*([`Unknown]*([`Unknown]*([`Univalued]* bool)))) t
  end

  val init : 'a -> ([`Unknown]*([`Unknown]*([`Unknown]*([`Unknown]*'a)))) t
  val get : ('a*('b*('c*('d*'e')))) t -> 'e

  val numerical : ('a*'x) t -> ([`Numerical]*'x) t
  val comparable : ('a*('b*'x)) t -> ('a*([`Comparable]*'x)) t
  val queryable :
    ('a*('b*('c*('d*'t')))) t -> ('a*('b*([`Queryable]*('d*'t')))) t
  val univalued :
    ('a*('b*('c*('d*'t')))) t -> ('a*('b*('c*([`Univalued]*'t')))) t
  val optionize : (('a*('b*'x)) t) option ->
    ('a*('b*([`Unknown]*([`Unknown]*( (('a*('b*'x)) t) option ))))) t

  (* phantom transformation function for various types *)
  val mkInt : int -> Types.phantom_int
  val mkString : string -> Types.phantom_string
  val mkBool : bool -> Types.phantom_bool

end = struct
  type 'a t = Obj.t

  (* phantom type definitions *)
  module Types = struct
    type phantom_int = ([`Numerical]*([`Comparable]*([`Unknown]*([`Univalued]* int)))) t
    type phantom_string = ([`Unknown]*([`Comparable]*([`Unknown]*([`Univalued]* string)))) t
    type phantom_bool = ([`Unknown]*([`Unknown]*([`Unknown]*([`Univalued]* bool)))) t

    (* we omit phantom type definition for other OCaml types *)
  end

  (* utility functions for phantom type transformation *)
  let init = fun x -> Obj.repr x
  let get = fun x -> Obj.obj x

  let numerical = fun x -> x
  let comparable = fun x -> x
  let queryable = fun x -> x
  let univalued = fun x -> x
  let optionize = fun x -> Obj.repr x

  let mkInt = fun i -> numerical (comparable (init i))
  let mkString = fun s -> comparable (init s)
  let mkBool = fun b -> init b

  (* we omit transformation function for other types *)

end
```

Figure 4.11: Phantom type module

```
(* Phantom functions for type checking *)
module PhantomFunctions = struct
  open Phantom (* module definition in Figure 4.11 *)

  (* query functions *)
  let from :
    ('a*('b*([`Queryable]*('d*'e)))) t -> ('a*('b*([`Queryable]*('d*'e)))) t =
    fun x -> x

  let join :
    ('a1*('b1*([`Queryable]*('d1*'e1)))) t ->
    ('a2*('b2*([`Queryable]*('d2*'e2)))) t ->
    (('a1*('b1*([`Queryable]*('d1*'e1)))) t -> ('a3*('b3*('c3*([`Univalued]*'e3)))) t) ->
    (('a2*('b2*([`Queryable]*('d2*'e2)))) t -> ('a3*('b3*('c3*([`Univalued]*'e3)))) t) ->
    unit = fun t1 t2 f1 f2 -> ()

  let where : ('ctx -> Types.phantom_bool) -> 'ctx -> 'ctx = fun f x -> x
  let having : ('ctx -> Types.phantom_bool) -> 'ctx -> 'ctx = fun f x -> x
  let limit : ('ctx -> Types.phantom_int) -> 'ctx -> 'ctx = fun f x -> x
  let offset : ('ctx -> Types.phantom_int) -> 'ctx -> 'ctx = fun f x -> x
  let orderby : ('ctx -> ('a * ([`Comparable] * ('c * ('d * 'e)))) t) -> 'ctx ->
              'ctx = fun f x -> x
  let groupby : ('ctx -> ('a * ('b * ('c * ([`Univalued] * 'e)))) t) -> 'ctx ->
              ('a * ('b * ('c * ([`Univalued] * 'e)))) t = fun f x -> f x
  let select : ('ctx -> 'a) -> 'ctx -> 'a = fun f x -> f x

  (* aggregate functions *)
  let count (xs: 'a collection) = Phantom.mkInt 0
  let sum (xs: (([`Numerical]*'others) Phantom.t) collection) = Collection.hd xs
  let avg (xs: (([`Numerical]*'others) Phantom.t) collection) = Collection.hd xs
  let min (xs: (('a*([`Comparable]*'others)) Phantom.t) collection) = Collection.hd xs
  let max (xs: (('a*([`Comparable]*'others)) Phantom.t) collection) = Collection.hd xs

  (* comparison operators: EQ/NE/LE/GE/LT/GT *)
  let ( = ) : ('a*([`Comparable]*'others)) Phantom.t -> ('a*([`Comparable]*'others)) Phantom.t ->
      Types.phantom_bool = fun a b -> Phantom.mkBool true

    (** logic operators: and, or, not *)
  let not : Types.phantom_bool -> Types.phantom_bool = fun a -> a
  let ( && ) : Types.phantom_bool -> Types.phantom_bool -> Types.phantom_bool = fun a b -> a

  (** Int: add, subtract, multiply and divide *)
  let ( + ) : Types.phantom_int -> Types.phantom_int -> Types.phantom_int = fun a b -> a
  let ( / ) : Types.phantom_int -> Types.phantom_int -> Types.phantom_int = fun a b -> a

  (** String: like *)
  let like : Types.phantom_string -> Types.phantom_string -> Types.phantom_bool =
     fun a b -> Phantom.mkBool true

  (* we omit arithmetic operations for Float, Int32, Numeric, Date, and Time, etc. *)
end
```

Figure 4.12: Shadow QQL Phantom Functions

4.5.1 Shadow orm objects

The basic idea of avatars is that for a query using normal orm objects and normal QQL built-in functions to be type safe if and only if the "phantomised" version of the query, i.e. the avatar, using the phantom versions of the orm objects and the phantom version of the built-in functions should be type safe. The point of this is that since the normal types of the elements involved cross two different systems and two different styles of computation, checking the normal types across the two systems is difficult to do at compile time, whereas the phantomised versions are straightforward to do at compile time, and, since we can encode these types in the OCaml type system, we can leave it to the OCaml compiler to do most of the work.

We have discussed the phantomised built-in functions previously. However, we have not yet introduced the phantomised orm objects.

We must first explain what a shadow orm object is. A shadow of each orm class is generated by the preprocessor. The issue is that we need to have an object that looks like a phantomised version of the original orm class, i.e. all the fields and methods of the original must be there with the same names, but all of those fields and methods must have the corresponding phantom types instead of the original normal types. We distinguish between a *shadow orm object* and a *phantom shadow orm object*. The former is a normally typed objects whose methods and fields have phantom types (but dummy implementations or values), while the latter is the former wrapped in a phantom type. Thus if `s` is a shadow orm object constructed from a normal orm object which has a field `t`, then we can get the appropriate phantom typed value of `t` with the expression `s#t`. However, the phantom shadow orm object corresponding to `s` has a phantom type, hence we cannot reference the `t` field from it directly. Instead we would first have to strip off the phantom type to get the proper `s` back first.

4.5.2 Example avatars for non-well typed queries

To help make the concept of type avatars clearer, we discuss two examples of type avatars that would be generated for incorrectly formed queries, and show how their type inconsistencies would cause the OCaml compiler to fail with a fatal error.

In our first example, we assume a program that deals with *user* and `order` orm classes. Suppose the programmer has entered the following, invalid QQL query:

```
Session.[select order from user]
```

The error is that order is not a field of the user class, but an entirely different class. During preprocessing of the program, the Qanat system would generate the following type avatar for this program, before passing the entire program, including generated code, to the OCaml compiler for compilation.

```
1  module Avatar : sig end = struct
2    let avatar user () =
3      let ctx = let user = PhantomFunctions.from (Session.user#phantom user) in
4                object
5                  method general = object method user = user end
6                  method aggregate = object end
7                end
8      in
9      let order = PhantomFunctions.select (fun ctx -> ctx#general#order) ctx in
10     (order)
11 end
```

We make the following observations about this avatar code:

1. The signature for the `Avatar` module is empty, thus none of the contents of the module are visible outside it, and, therefore, none of its methods can ever be executed.

2. The module contains a function, `avatar`, that takes a parameter called `user` and a unit. The `user` parameter is simply to provide a value for the queryable object in the query, i.e. a

user object, as that is what appears in the `from` clause. If there were more objects in the `from` clause, more parameters would be required. The unit is there in case the `from` clause is empty.

3. Lines 3 to 7 create a `context`. This is an anonymous object, containing two methods, one for the *general* context part, and one for the *aggregate* context part. In this case, there is no `group-by` clause or aggregate functions being used in the query so we do not need to generate the special aggregate part of the context and therefore leave it empty.

4. A context part is a constant function that returns an anonymous object with one constant method for each queryable object that can be used as the root of expressions in the rest of the query.

5. The name of each of these methods is the name of the orm class of the queryable object, and the return value is the queryable phantom shadow of that orm class.

6. Line 3 constructs this phantom value: it applies `Session.user#phantom` to the parameter variable `user`, and returns a value (the actual value does not matter) which has the phantom type of objects of the orm `user` class.

7. Line 3 also applies `PhantomFunctions.from` to this phantom value. This function, defined in Fig. 4.12, is a function that maps queryable phantom types to queryable phantom types. Again the actual return value does not matter: the important point is that this line will not type check unless `Session.user#phantom user` is of a queryable phantom type. That will only be the case if the user class is an orm class.

8. Thus the type of the return value of the `user` context method in line 5, is that of a phantom value obtained from the `user` orm class.

9. Line 10, then, gives a phantom typed return value of the query: in this case a tuple containing a single value called `order`.

10. Line 9 obtains the value for this query return value. `PhantomFunctions.select` (c.f. Fig. 4.12) is of type `('ctx ->'a) -> 'ctx ->'a`, so, for this line to type check, the general part of the context object must have a method called `order`. In this case there is none, so the OCaml type checker fails and a compile-time fatal error occurs as required.

For our second error case example, we take the following query where the `user` orm class has a `string` type `email` field, but the programmer has tried to compare this to the integer `200` in the `where` clause.

```
Session.[from user where user#email > 200]
```

The resulting generated avatar is as follows

```
1  module Avatar : sig end = struct
2    let avatar user () =
3      let ctx = let user = PhantomFunctions.from (Session.user#phantom user) in
4                object
5                  method general = object method user = user end
6                  method aggregate = object end
7                end
8      in
9      let ctx = PhantomFunctions.where
10         (fun ctx -> (Phantom.get ctx#general#user)#email
11                     PhantomFunctions.(>)
12                     (Phantom.mkInt 200)) ctx
13     in
14     let user = PhantomFunctions.select (fun ctx -> ctx#general#user) ctx in
15     (user)
16 end
```

Much of the analysis is the same as in the previous case. Here we will concentrate on the differences.

1. Lines 9 to 12 appear to modify the `context`. However, since we do not care about the actual values, the real work is happening in the types, and the type of the return from `PhantomFunctions.where` is exactly the same as the type of the `'ctx` parameter.

2. `PhantomFunctions.where` is of type `('ctx -> Types.phantom_bool) -> 'ctx -> 'ctx`. Hence its first argument must be a function from its second parameter to a phantom bool. Thus the body of the first (function) parameter must have type phantom bool to type check correctly.

3. Line 12 is lifting the literal `200` to the phantom type for integers

4. `ctx#general#user` is a phantom shadow of a `user` object. We need its `email` field so we first need to strip off the object's phantom type with `Phantom.get` to get a normal shadow object. Then we can get the email field, which, since this came from a shadow user object will have the phantom version of the type of this field in the real orm `user` object, i.e. a phantom string.

5. Now the type checking fails since the phantom comparison operator requires the types of its arguments to be both comparable types (no problem yet) and the same (hence the failure).

4.5.3 Avatar generation

If queries are syntactically correct, preprocessing is performed before the generation of avatar data in order to ensure that each query source (i.e. orm class) has a unique alias in the query, and that there are no free identifiers in the query (i.e. all identifiers are declared either as orm classes or query variables). Moreover, a new query clause, `group by default`, is introduced during preprocessing when an aggregate function is used in the query without an explicit group-by clause. This default group-by does not change the semantics of query. It is merely used to unify the aggregate operations by managing the change of query context (making the aggregate function available for use in the rest of the query).

`select count` e_1`, max` e_2 `from x where` $e \rightsquigarrow$
`select count` e_1`, max` e_2 `from x as x group by default where` e

Finally, if the query had no explicit `select` clause, a default one is added that simply lists all the elements of the `from` clause, i.e. the equivalent of "`select *`".

Avatars are generated by translating queries into the application of phantom query functions. Expressions, like field access and boolean predicates, are transformed into phantom expressions. The translation is performed in a specific order, starting from the `from` clause, followed by `where`, `group-by`, `order-by`, etc, ended with `select`. These expressions are chained together to model type constraints of the query.

Since we already have shadow phantom functions for all QQL query constructs, the generation of an avatar is, in principle, straightforward: we walk the QQL abstract syntax tree and translate each QQL query clause into an application of the corresponding shadow function, and wire the calls together so that return types are required to appropriately match the caller's expectations. To do this, we need to be able to apply the semantic constraints of QQL as to what expressions are allowed in different parts of the full query expression, e.g. that if we refer to an orm class in the `select`, clause, that the class appears in the `from` clause.

The *context* at any location in a query expression is the collection of queryable elements that can be referred to at that point in the expression. A queryable element is either an orm class or a variable of query type.

The context varies depending on

1. What clause of the query the location is in, e.g. the `where` clause can always refer to any queryable element, the `having` clause can only refer to queryable elements listed in the `group by` clause.

2. Whether the query has a `group-by` clause or an aggregate function, e.g. any queryable element can be referred to in the `select` clause if there is no `group-by` clause or aggregate function in the query. Otherwise, outside the argument of an aggregate function, it can only refer to the queryable elements listed in the `group by` clause.

3. if the location is in an argument to an aggregate function, then any queryable element can be referenced.

To capture the rules involved, we designed a two part context which can be modified depending on which clauses appear in the query.

- The *general* sub-context contains only instances that can be referred to *outside* the arguments to aggregate functions. If there is no `group-by` in the query, this includes all the classes referred to in the `from clause`. If there is a `group-by` clause, this is further restricted to the instances that are referred to in that clause.

- The *aggregate* sub-context contains only instances that can be referred to as a parameter of an aggregate function. These are all the query sources.

The context is not a usual container data structure. It is never instantiated at all and therefore never holds any data: it is its type which contains the information required, thus, rather than being a container type, one can think of a context as a *type container*.

In order to get a generic query context structure that is suitable for queries defined based on any kind and any number of queryable sources (i.e. the orm classes and sub-queries), the query context structure is designed as anonymous objects [CMP00]. OCaml has various structures to contain data, but it requires explicitly defining the type of all these structures before using them, except for the anonymous object. Anonymous objects are similar to standard object-oriented classes, but can be used directly in a way similar to tuple values. However, unlike tuples, we can define labels and methods for each encapsulated value in an anonymous object.

Therefore, in an avatar, we construct an initial context object from the `from` clause, and pass it on to each shadow query function in turn. Some clauses, such as `where` do not change the query context. Therefore their return context value is of the same type as their input context value. Hence they do not change the context. On the other hand, clauses like `group-by` do change the query context, in this case, a new context object type is constructed based on the old context object type and the fields/entities used in the `group-by` clause. Specifically, the changes in the new context is that: the *aggregate* sub-context is reset to support using aggregate functions, i.e. this sub-context includes all query sources as well those instances used for group partition and is represented in the form of a collection (aggregate functions are defined on collection values). Meanwhile, the *general* sub-context is restricted to only include instances that are used for group partitioning, this is based on the fact that such queries can only return field values that are used in the group partitioning. Then, clauses following the group-by will be defined based on the new query context, which now is enabled to use aggregate functions on arbitrary query sources.

From this, it is clear that the order in which the shadow functions for the clauses appear is critical: `from` must appear first to set up the initial context. `where` must come next as it uses the initial context unchanged. `group by` must come next as it will change the context and all the remaining clauses must use the new context it creates. `having`, `order by`, `offset` and `limit` can then appear in any order as none of them change the context. `select` comes last as that is what sets up the final return value of the query (and it also uses the context modified by the `group by`.

We present an example to show the construction and modification of the query context object by the `from` and `group-by` clauses. Consider the following query (the `select` clause is omitted):

```
from employee as emp group by emp#working_years as wy
```

We start by building a query context instance from the `from` clause.

```
let ctx =
  (* phantom transformation for employee entity *)
  let emp = from (Session.employee#phantom employee) in
  object
```

```
    (* methods to return the general and aggregate sub-context *)
    method general = object
                       method emp = emp
                     end
    method aggregate = object end
  end
```

The first step is perform the phantom transformation for entity employee, the transformation function is pre-generated in the Session module (see section 4.8). The application of the shadow phantom function `from` to the phantom shadow employee object will only type check correctly if the phantom shadow employee is a queryable type. The type of its return value is the same as that of its argument. The transformed `employee` is bound to its alias `emp`, which is then included in the *general* sub-context of the returned context object (an anonymous object). The *general* sub-context is also an anonymous object, inside which the query sources (i.e. `emp` in this example) can be obtained by invoking the method of the same name, which returns the `emp` value. Note that in the method definition, the `emp` on the leftside of `=` is the method name, and the one on the rightside is the `emp` value. A method `m` of an object `obj` can be referred to by the form `obj#m`.

Having dealt with the from clause and built our initial context, we now handle the `group by` clause and build the new query context.

```
let ctx =
  (* obtain working_years of the employee based on the old context *)
  let wy = groupby (fun ctx -> (Phantom.get ctx#general#emp)#working_years) ctx in
  (* a new context instance is created *)
  object
    method general = object
                       method wy = wy
                     end
    method aggregate =
           new collection [(object
                             method wy = wy
                             method emp = ctx#general#emp
                           end)]
  end
```

Firstly, the type checking is guaranteed by applying shadow function `groupby`, which ensures that only univalued values are used in the `group-by` clause. Function `group-by` takes two parameters, the first one is a function for obtaining the value used for group allocation from a query context object; the second one is the context instance returned from clauses that were defined before `group-by`. Since `emp` is a phantom-type entity now, it is necessary to convert `emp` back to non-phantom shadow object, on which the primitive function (`#`) can be used to return the value of an object field. Because the object is a shadow entity rather than a normal one, the field returned will be a phantom type value. Then a new context instance is initialised, inside which the *general* sub-context is restricted to only contain the field used for group allocation, i.e. `wy` in this example; the *aggregate* sub-context is set to contain all values that can be referred to in the aggregate expression, i.e. field `wy` defined in this `let` block, and entity `emp` obtained from the old context instance. Note that the *aggregate* method in the new context instance returns a collection type. The reason for this is that aggregate functions are applied to collection data only. While the old *aggregate* sub-context is an empty object, thus invoking aggregate functions in clauses before the `group-by` (in this case we refer to the order of phantom shadow functions in the query object, not the order of clauses in the text of the QQL query) will cause a compile-time error. The construction of a new query context instance means that both *general* and *aggregate* sub-contexts are changed from the content and the type, compared to the old context instance.

Note that this redefinition of the context type is legal in OCaml, because anonymous objects do not have a pre-defined type. They can be created to contain arbitrary fields and methods.

Figure 4.13 and 4.14 define the translation of QQL queries into type avatars. A`Jctx, exprK` denotes the translation of expression `expr` under a specified query context, which consists of sub-contexts for ordinary calculation and aggregation respectively. A′`Jctx, aeK` denotes the translation of field access expression `ae`. Symbol `$expr$` indicates meta action during the translation; `infer_pf`

performs the action of inferring a phantom transformation function for a query variable. The avatar generation rules in Figure 4.13 includes four parts. These rules are applied in a nested style, starting from translating the query clauses.

- the first part (i.e. $\mathrm{A}'\mathrm{J}ctx, ae\mathrm{K}$) translates standard field access expressions into the expression for obtaining field value from a phantom-type data based on the given context instance, either *general* or *aggregate* sub-context.

  Because composition types like records and objects are encapsulated as phantom data, standard access operators cannot work on these data. Thus additional inverse action from phantom data to composition data is performed through the application of `Phantom.get`.

- the second part translates standard query expressions into the expression for applying the corresponding predefined phantom functions or for converting standard values into phantom type.

  Note that aggregate expressions in QQL can be expressed in two forms, one is the standard aggregate function invocation based on `group-by` clause, another one is to invoke aggregate functions as a method belonging to collection-type data. The translations are different. The first one is translated based on the *aggregate* sub-context, while the latter one is based on the *general* sub-context. The aggregate method can be invoked on any collection-type data without requiring a `group-by` clause.

  The translation of constant values and variables generates an expression for transforming these data into its phantom type form. For variables, it is necessary to perform type inference to get its phantom transformation function, c.f. section 4.7 for detail.

  There is one complication here, also discussed in detail section 4.7. Some constructs possible in a query, e.g. a `where` condition that compares two variables to each other where those variables do not appear in any other part of the query, impose only a polymorphic type constraint on the variables (namely, that they must be of comparable type) but do not impose any specific type on them. In such cases there is no suitable phantom function to generate or find for these variables and, instead, we replace the expression that imposes the polymorphic constraint in the QQL query with one that will impose the same polymorphic constraint in the avatar. It should be noted, therefore, that the cases that can lead to such a polymorphic constraint are very limited. In particular, if an expression of the form $e_1\, op\, e_2$ refers to a variable identified, by the variable type constraint inference procedure described in section 4.7, as polymorphic, then both e_1 and e_2 must be simple variable names and *op* must be a comparison operation, and hence the result of the operation must be of type *bool*. If any of these requirements are not satisfied, the variables involved would not be polymorphic. Similar observations about the other cases apply and explain the very specific nature of the corresponding manipulations in the avatar generation rules.

- The third part shows how to translate QQL clauses into the application of shadow phantom query functions, generating or passing the query context instance for following query clauses. Since, for each QQL clause, we predefined a shadow phantom function for type checking, these translations are simple.

- The last part defines how to generate the expression for transforming queryable orm entities and sub-queries (defined as query variables in the `from` clause) into phantom-type values. Each query object, after compilation, includes phantom transformation functions for its return value, and an encapsulated `select` field (see section 4.8). We pre-generate phantom transformation functions for each orm class in the `Session` module, and the expression for performing translation can be generated straightforwardly.

  During this process, we add parameters to the generated avatar function. For sub-query variables, one parameter is added representing the return value of the query-type variable. For orm class expressions, one parameter is added for the orm class.

In practice, a type avatar is a parameterised function specific to a query. It takes parameters representing the orm classes and the return value of query-type variables referred to in the query.

```
Field access expression (ae)
  A′J∅, idK ⇝ id
  A′Jctx, idK ⇝ ctx#id
  A′Jctx, ae#fK ⇝ (Phantom.get A′Jctx, aeK)#f
  A′Jctx, ae.fK ⇝ (Phantom.get A′Jctx, aeK).f
  A′Jctx, ae#[x => e]K ⇝ map (λx -> AJ∅, eK) A′Jctx, aeK
  A′Jctx, valof (ae)K ⇝ PhantomFunctions.valof A′Jctx, aeK
  A′Jctx, itemsof (ae)K ⇝ PhantomFunctions.itemsof A′Jctx, aeK

Expression (e)
  AJctx, cK ⇝ Phantom.mkT c (* suppose constant c is of type t *)
  AJctx, :vK ⇝ $infer_pf v$ v
  AJctx, aeK ⇝ A′Jctx#general, aeK
  AJctx, ae#aggr_function()K ⇝ PhantomFunctions.aggr_function A′Jctx#general, aeK
  AJctx, aggr_function eK ⇝
            PhantomFunctions.aggr_function (map (λx -> AJx, eK) ctx#aggregate
  AJctx, [e1;...;en]K ⇝ new collection [(AJctx, e1K;...;AJctx, enK)]
  AJctx, e1 op e2K ⇝
    when e1 and e2 contain polymorphic variables, the generated expression is:
            Phantom.mkBool (e1 op e2)
    otherwise,
            PhantomFunctions.(op) AJctx, e1K AJctx, e2K
  AJctx, e1 in e2K ⇝
    when e1 and e2 contain polymorphic variables, the generated expression is:
            Phantom.mkBool (Collection.mem e1 e2)
    otherwise,
            PhantomFunctions.in AJctx, e1K AJctx, e2K
  AJctx, e is [not]^0.1 Null K ⇝
    when e is a variable, the generated expression is:
            Phantom.mkBool (Library.isSome e)
    otherwise,
            PhantomFunctions.isSome AJctx, eK
  AJctx, e1 between e2 and e3K ⇝
    when e1, e2 or e3 contain polymorphic variables, the generated expression is:
            Phantom.mkBool (Library.between e1 (e2, e3))
    otherwise,
            PhantomFunctions.between AJctx, e1K (AJctx, e2K, AJctx, e3K)
```

Figure 4.13: Translation of QQL queries into type avatars

Because type checking is performed under phantom types, these parameters need to be transformed into phantom data. ORM entities can be transformed by applying functions pre-generated in the session module. Query variables need additional type inference (not full type inference, but only up to type equivalences) to identify the phantom transformation function (we present a modified Hindley-Milner algorithm in section 4.7 for this purpose). The content of the avatar function is generated from query clauses. Clauses like `where`, `having`, `order-by`, `limit` and `offset` are mapped in a straightforward manner to the application of corresponding phantom functions, which take arguments translated from the clause expression. Clauses `from` and `group-by` define query contexts (including the *general* sub-context for general expressions and the *aggregate* sub-context for aggregation expression), which are constructed from query sources. The sub-context for aggregation remains empty until a `group-by` clause is encountered. The `group-by` clause rebuilds the query context, which supports the use of aggregate expressions in the following clauses.

For concreteness, we show an example of an avatar generation from a QQL query. Consider a query that, given a selection, `q`, of departments that are returned from another query, finds the employees in those departments who have been working in the company for less than `ny` years but who have worked on more than `np` projects.

```
Query clause (q)
  AJ∅, from f1 as id1 [join f2 as id2 on id1#e1=id2#e2]*, ..., fn as idnK ⇝
           let ctx = let id1 = PhantomFunctions.from (AJ∅, f1K) in
                     let id2 = PhantomFunctions.from (AJ∅, f2K) in
                     PhantomFunctions.join id1 id2
                          (λid1 -> A'J∅, id1#e1K) (λid2 -> A'J∅, id2#e2K);
                     ...
                     let idn = PhantomFunctions.from (AJ∅, fnK) in
                     object
                       method general =
                         object method id1 = id1;; ... method idn = idn;; end
                       method aggregate = object end
                     end
  AJctx, where eK ⇝ PhantomFunctions.where (λx -> AJx, eK) ctx
  AJctx, group by defaultK ⇝ let ctx = object
                                 method general = object end
                                 method aggregate = new collection [(ctx#general)]
                               end
  AJctx, group by ae1 as k1, ..., aen as knK ⇝
           let ctx = let k1 = PhantomFunctions.groupby (λx -> AJx, ae1K) ctx in
                     ...
                     let kn = PhantomFunctions.groupby (λx -> AJx, aenK) ctx in
                     object
                       method general =
                         object method k1 = k1;; ... method kn = kn;; end
                         method aggregate = new collection [(
                            object
                                 inherit methods from ctx#general;;
                                 method k1 = k1;; ... method kn = kn;;
                             end)]
                     end
  AJctx, having eK ⇝ PhantomFunctions.having (λx -> AJx, eK) ctx
  AJctx, order by e1 dir, ..., en dirK ⇝
                     PhantomFunctions.orderby (λx -> AJx, e1K) ctx;
                     ...
                     PhantomFunctions.orderby (λx -> AJx, enK) ctx;
  AJctx, limit eK ⇝ PhantomFunctions.limit (λx -> AJx, eK) ctx
  AJctx, offset eK ⇝ PhantomFunctions.offset (λx -> AJx, eK) ctx
  AJctx, select e1 as id1, ..., en as idnK ⇝
                     let id1 = PhantomFunctions.select (λx -> AJx, e1K) ctx in
                     ...
                     let idn = PhantomFunctions.select (λx -> AJx, enK) ctx in
                     (id1, ..., idn)
From expression (f)
  AJ∅, :qK ⇝ (* add a variable q_result as avatar parameter *)
                     q#phantom_transform_context#phantom q_result
  AJ∅, aeK ⇝ (* add free identifier in ae as avatar parameter, suppose it is clazz_a *)
                     let clazz_a = Session.clazz_a#phantom clazz_a in A'J∅, aeK
```

Figure 4.14: Translation of QQL queries into type avatars (cont.)

This query involves orm class `employee`, variable `q` denoting a sub-query that filters out a group of departments, and variables `ny` and `np`, representing the specified number of years and of projects.

```
Session.[select emp#email as email, q#dept#name as deptname
          from employee as emp join :q on emp#department#id = q#dept#id
          where emp#working_years < :ny and emp#projects#count () > :np]
```

This query uses an explicit join over class `employee` and sub-query `q`. The corresponding avatar function takes two arguments, one for `employee` and one for variable `q`. The variable for `q` represents the return value of this sub-query. Query object `q` contains the expression of the query represented as an abstract syntax tree and functions related to the phantom transformation of the query result used in the avatar, and result value construction to be used during the query execution.

Since type checking in an avatar is based on phantom-typed values, the first step of avatar generation is to obtain or infer the phantom transformation function for each avatar function parameter. Variable `q` is a sub query, thus the phantom transformation function for the query return value can be directly obtained from the sub-query object itself (each query object after compilation contains functions for the phantom transformation of the query return value and result construction, c.f. section 4.8). Based on the type inference algorithm proposed in section 4.7, it is known that the type of variable `ny` has to be the same as that of `employee`'s `working_years`, and that the type of variable `np` is `int`. Thus the phantom translation of `ny` can be performed by applying the pre-generated function for field `employee.working_years`, and np can be transformed using function `Phantom.mkInt`. The phantom transformation of these values can be defined as follows:

```
employee ⇝ Session.employee#phantom employee
q_result ⇝ q#phantom_transform_context#phantom q_result
ny ⇝ Session.employee#working_years#phantom ny
np ⇝ Phantom.mkInt np
```

Once we get these phantom transformation functions, the avatar function is generated as follows; each query clause, such as `from`, `where`, `group-by`, etc, is mapped to an application of a shadow query function, the parameter of which is constructed from the clause expression based on the query context instance. The query context instance is constructed from clause `from`, and is kept until construction of the avatar function finishes. Clauses like `where`, `order-by`, `having`, `limit` and `offset` pass the context instance without modification, until meeting clause `group-by`, which modifies the context instance in order to support aggregate functions. The expression used as a parameter of an aggregate function is constructed from the *aggregate* sub-context, while other expressions are constructed from the *general* sub-context. Since values in avatar are of phantom types, standard field access operators (e.g. "`#`" for accessing an object field and "`.`" for accessing record fields) become invalid. To get these operators working, a inverse action is performed to convert these values back into standard objects or records that contains phantom-typed fields. For example, expression `emp#working_years` is mapped to `(Phantom.get ctx#general#emp)#working_years`. Expression `emp#projects#count ()` does not rely on the `group-by` clause to perform the aggregate function, it is treated the same as an ordinary field access expression and is constructed based on the *general* sub-context. Figure 4.15 shows the type avatar generated for the above query. Note that the avatar function is hidden inside a local module with an empty signature and therefore is unavailable to the application programmer.

We present another example of avatar generation in Figure 4.16, demonstrating the generation of an avatar when using aggregate functions and a group-by clause. Suppose, in a bookshop example, each customer is associated with an role (either personal buyer or company buyer) indicating the different discounts that customers gets. The query splits all customers with more than 10 orders into one of these two roles and returns the number of these customers in each role:

```
Session.[select role#name as rolename, (count cust) as count
          from customer as cust
          where cust#orders#count() > 10
          group by cust#role as role]
```

This query involves a field-access-expression based aggregate method and an explicit group-by aggregate function. The avatar module is generated with an empty signature, ensuring that avatar functions can never be invoked by users.

```
let query ~q ~ny ~np =
  (* Generated avatar module with empty signature *)
  let module Avatar : sig end = struct
     open PhantomFunction (* overloaded QQL functions *)

     let avatar employee q_result () =
       (* build context instance from the from clause *)
       let ctx = let emp = from (Session.employee#phantom employee) in
                 let q = from (q#phantom_transform_context#phantom q_result) in
                 join emp q
                     (fun emp -> (Phantom.get (Phantom.get emp)#department)#id)
                     (fun q -> (Phantom.get (Phantom.get q)#dept)#id);
                 object
                   method general =
                       object method emp = emp;; method q = q;; end
                   method aggregate = object end
                 end
       in
       (* where condition *)
       let ctx = where (fun ctx ->
            ((Phantom.get ctx#general#emp)#working_years <
               (Session.employee#working_years#phantom ny))
            and ((count (Phantom.get ctx#general#emp)#projects) > (Phantom.mkInt np)) ) ctx
       in
       (* select result *)
       let email = select (fun ctx -> (Phantom.get ctx#general#emp)#email) ctx in
       let deptname = select (fun ctx ->
                              (Phantom.get (Phantom.get ctx#general#q)#dept)#name) ctx in
       (email, deptname)
    end
  in
  (* we omit other parts of the query *)
```

Figure 4.15: Type avatar example

As in our previous example (Figure 4.15), the application of shadow query functions is chained together by building, passing and modifying the query context instance. The `from` clause initialises a query context, which contains query source `cust` in the *general* sub-context, and leaves the *aggregate* sub-context empty indicating that aggregate functions can not be used. The `where` clause is mapped to the application of shadow function `where`, the parameter of which is built based on the *general* sub-context. Function `where` doesn't change the query context, it checks that the argument condition results in a boolean value and then returns the original query context instance. The `group-by` clause partitions records into groups, in this case, the query context instance is reset to include a *general* sub-context which only contains the fields referred to in the group-by clause, and an *aggregate* sub-context which contains all query sources as well as group-by criteria fields. Since the *aggregate* sub-context is no longer empty, it is now possible to define aggregate expressions in the following query clauses (i.e. `having` and `select`). In this example, we use `count` in the `select` clause.

```
let query =
  (* Generated avatar module with empty signature *)
  let module Avatar : sig end = struct
     open PhantomFunction (* overload QQL functions *)

     let avatar customer () =
       (* build context instance from the from clause *)
       let ctx = let cust = from (Session.customer#phantom customer) in
                 object
                   method general =
                       object
                              method cust = cust
                       end
                   method aggregate = object end
                 end
       in
       (* where condition based on the general subcontext *)
       let ctx = where (fun ctx ->
                   (count (Phantom.get ctx#general#cust)#orders) > (Phantom.mkInt 10) ) ctx
       in
       (* new context instance constructed from group-by *)
       let ctx = let role = groupby (fun ctx -> (Phantom.get ctx#general#cust)#role) ctx in
                 object
                   method general = object method role = role end
                   method aggregate =
                       new collection [(object method role = role
                                               method cust = ctx#general#cust
                                        end)]
                 end
       in

       (* select result *)
       let rolename = select (fun ctx -> (Phantom.get ctx#general#role)#name) ctx in
       let count = select (fun ctx -> count (map (fun x -> x#cust) ctx#aggregate)) ctx in

       (* query return *)
       (rolename, count)
  end
in
(* we omit other parts of the query *)
```

Figure 4.16: Another example of a type avatar

4.6 Compile-time well-typing of QQL queries

Because of the support for overloaded functions in QQL, necessitated by the nature of the underlying SQL database, the OCaml compiler is incapable of performing type checking over QQL queries using the QQL's natural type system. Rather than proposing a standalone type checking algorithm for an extension to OCaml that encompasses the necessary QQL types, we pre-generate avatar data for each query under the QQL phantom typing system, which provides the opportunity of leveraging the standard OCaml compiler to check that the queries are well-typed. This means that the compile-time type checking is performed on the phantom typing system rather than standard QQL typing system.

First, note that the phantom type system is, by construction, type checkable at compile time by the OCaml compiler. Hence we need only to show that a query is well-typed in the QQL type system, $\vdash$ if and only if it is well typed in the phantom type system, $\vdash_p$, and, furthermore, the same query expressions in corresponding environments have corresponding types, where a QQL type corresponds to a phantom type if the latter is obtained via the phantom type transform from the former and vice versa. This final condition ensures that a trivial phantom type transformation (for example, to a single type) will not satisfy the requirements and means that we need the phantom type transform to be invertible.

Formally, we extend the definition of $\mathrm{P}\,\mathrm{J}\cdot\mathrm{K}$ to type environments: let Ξ be a type environment in the QQL type system, then we define $\mathrm{P}\,\mathrm{J}\Xi\mathrm{K}$ to be the type environment in the phantom QQL type system that has the same domain as Ξ, but in which type τ_p, associated with *identifier* n, is the phantom type of the the QQL type τ associated with n in Ξ. Even with this extension, Lemma 4.3, which guarantees the invertibility of the phantom type transform, still holds, as we have just extended the type system of QQL and phantom QQL by a new composite type, and hence the reasoning that was used in the proof of the lemma still holds. Further, this will not cause any problems for our analysis as the new type can never appear to the right of a type judgement.

Theorem 4.1 *Given query q and QQL schema environment S, QQL variable definition environment D, phantom QQL schema environment S', and phantom QQL variable definition environment D', then*

1. $\mathrm{S}; \mathrm{D}; \emptyset \vdash \mathtt{q} : \tau \Rightarrow \mathrm{P}\,\mathrm{J}\mathrm{S}\mathrm{K}; \mathrm{P}\,\mathrm{J}\mathrm{D}\mathrm{K}; \emptyset \vdash_p \mathtt{q} : \mathrm{P}\,\mathrm{J}\tau\mathrm{K}$
2. $\mathrm{S}'; \mathrm{D}'; \emptyset \vdash_p \mathtt{q} : \tau' \Rightarrow \mathrm{P}^{-1}\,\mathrm{J}\mathrm{S}'\mathrm{K}; \mathrm{P}^{-1}\,\mathrm{J}\mathrm{D}'\mathrm{K}; \emptyset \vdash \mathtt{q} : \mathrm{P}^{-1}\,\mathrm{J}\tau'\mathrm{K}$

The proof is by inspecting every matching pair of typing rules (R, R') from the QQL and the phantom rule set. Given that $\mathrm{P}\,\mathrm{J}\cdot\mathrm{K}$ defines a bijective mapping between the QQL types and the phantom QQL types, we need only check that the structure of R and R' are the same and that, for corresponding environments (S, D, Γ) and (S', D', Γ') where $S' = \mathrm{P}\,\mathrm{J}S\mathrm{K} \wedge D' = \mathrm{P}\,\mathrm{J}D\mathrm{K} \wedge \Gamma' = \mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}$, then the types in corresponding locations in the structures of R and R' are also related by $\mathrm{P}\,\mathrm{J}\cdot\mathrm{K}$. Finally a simple inductive argument delivers the required result.

We show a few cases of the proof, such as basic typing rules for *integer literal*, *record field access*, and *like*. Other cases can be constructed in a similar manner.

Case - Integer Literal

$$\frac{}{\mathrm{S}; \mathrm{D}; \Gamma \vdash \mathtt{i} : \mathit{int}}\ \textsc{Lit-int}$$

$$\frac{}{\mathrm{S}'; \mathrm{D}'; \Gamma' \vdash_p \mathtt{i} : \mathrm{P}\,\mathrm{J}\mathit{int}\mathrm{K}}\ \textsc{pLit-int}$$

Given the bijective mapping function $\mathrm{P}\,\mathrm{J}\cdot\mathrm{K}$ between the QQL types and the phantom QQL types, the typing rule `LIT-INT` is mapped to,

$$\frac{}{\mathrm{P}\,\mathrm{J}\mathrm{S}\mathrm{K}; \mathrm{P}\,\mathrm{J}\mathrm{D}\mathrm{K}; \mathrm{P}\,\mathrm{J}\Gamma\mathrm{K} \vdash_p \mathtt{i} : \mathrm{P}\,\mathrm{J}\mathit{int}\mathrm{K}}$$

which is equivalent to the phantom typing rule `pLIT-INT`, given that $S' = \mathrm{P}\,\mathrm{J}S\mathrm{K} \wedge D' = \mathrm{P}\,\mathrm{J}D\mathrm{K} \wedge \Gamma' = \mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}$.

Similarly, ᵖLIT-INT is mapped to a corresponding typing rule

$$\frac{}{\mathrm{P}^{-1}\,\mathrm{J}S'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}D'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}\Gamma'\mathrm{K}\vdash \mathtt{i} : \mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}\mathit{int}\mathrm{K}\mathrm{K}}$$

Since $\mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}\tau\mathrm{K}\mathrm{K} = \tau$ and $S' = \mathrm{P}\,\mathrm{J}S\mathrm{K} \wedge D' = \mathrm{P}\,\mathrm{J}D\mathrm{K} \wedge \Gamma' = \mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}$, the above typing rule is equivalent to typing rule LIT-INT. Thus, we get that the typing rules LIT-INT and ᵖLIT-INT are images of each other under $\mathrm{P}\,\mathrm{J}\cdot\mathrm{K}$ and $\mathrm{P}^{-1}\,\mathrm{J}\cdot\mathrm{K}$ respectively.

Case - Record Field Access

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{x} : \{r_1 : t_1;\cdots;r_n : t_n\} \qquad 1 \le \mathtt{i} \le \mathtt{n}}{\mathrm{S};\mathrm{D};\Gamma\vdash \mathtt{x}.r_i : t_i}\ \textsc{record field access}$$

$$\frac{\mathrm{S}';\mathrm{D}';\Gamma'\vdash_p \mathtt{x} : \mathrm{P}\,\mathrm{J}\{r_1 : t_1;\cdots;r_n : t_n\}\mathrm{K} \qquad 1 \le i \le n}{\mathrm{S}';\mathrm{D}';\Gamma'\vdash_p \mathtt{x}.r_i : \mathrm{P}\,\mathrm{J}t_i\mathrm{K}}\ \textsc{p}\textsc{Record field access}$$

The use of the phantom transformation function, $\mathrm{P}\,\mathrm{J}\cdot\mathrm{K}$, yields that the typing rule, RECORD FIELD ACCESS, is mapped to a rule equivalent to ᵖRECORD FIELD ACCESS.

$$\frac{\mathrm{P}\,\mathrm{J}S\mathrm{K};\mathrm{P}\,\mathrm{J}D\mathrm{K};\mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}\vdash_p \mathtt{x} : \mathrm{P}\,\mathrm{J}\{r_1 : t_1;\cdots;r_n : t_n\}\mathrm{K} \qquad 1 \le \mathtt{i} \le \mathtt{n}}{\mathrm{P}\,\mathrm{J}S\mathrm{K};\mathrm{P}\,\mathrm{J}D\mathrm{K};\mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}\vdash_p \mathtt{x}.r_i : \mathrm{P}\,\mathrm{J}t_i\mathrm{K}}$$

and ᵖRECORD FIELD ACCESS is mapped to a rule equivalent to RECORD FIELD ACCESS. The desired result is immediate.

$$\frac{\mathrm{P}^{-1}\,\mathrm{J}S'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}D'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}\Gamma'\mathrm{K}\vdash \mathtt{x} : \mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}\{r_1 : t_1;\cdots;r_n : t_n\}\mathrm{K}\mathrm{K} \qquad 1 \le i \le n}{\mathrm{P}^{-1}\,\mathrm{J}S'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}D'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}\Gamma'\mathrm{K}\vdash \mathtt{x}.r_i : \mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}t_i\mathrm{K}\mathrm{K}}$$

Case - Like

$$\frac{\mathrm{S};\mathrm{D};\Gamma\vdash e_1 : \mathit{string} \qquad \mathrm{S};\mathrm{D};\Gamma\vdash e_2 : \mathit{string}}{\mathrm{S};\mathrm{D};\Gamma\vdash e_1\ \mathtt{like}\ e_2 : \mathit{bool}}\ \textsc{Like}$$

$$\frac{\mathrm{S}';\mathrm{D}';\Gamma'\vdash_p e_1 : \mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K} \qquad \mathrm{S}';\mathrm{D}';\Gamma'\vdash_p e_2 : \mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K}}{\mathrm{S}';\mathrm{D}';\Gamma'\vdash_p e_1\ \mathtt{like}\ e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}\ \textsc{p}\textsc{Like}$$

Similarly, typing rule LIKE is mapped to

$$\frac{\mathrm{P}\,\mathrm{J}S\mathrm{K};\mathrm{P}\,\mathrm{J}D\mathrm{K};\mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}\vdash_p e_1 : \mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K} \qquad \mathrm{P}\,\mathrm{J}S\mathrm{K};\mathrm{P}\,\mathrm{J}D\mathrm{K};\mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}\vdash_p e_2 : \mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K}}{\mathrm{P}\,\mathrm{J}S\mathrm{K};\mathrm{P}\,\mathrm{J}D\mathrm{K};\mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}\vdash_p e_1\ \mathtt{like}\ e_2 : \mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}}$$

which is equivalent to ᵖLIKE, given that $S' = \mathrm{P}\,\mathrm{J}S\mathrm{K} \wedge D' = \mathrm{P}\,\mathrm{J}D\mathrm{K} \wedge \Gamma' = \mathrm{P}\,\mathrm{J}\Gamma\mathrm{K}$.

While, ᵖLIKE is mapped to a standard typing rule equivalent to LIKE.

$$\frac{\begin{array}{c}\mathrm{P}^{-1}\,\mathrm{J}S'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}D'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}\Gamma'\mathrm{K}\vdash e_1 : \mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K}\mathrm{K}\\ \mathrm{P}^{-1}\,\mathrm{J}S'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}D'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}\Gamma'\mathrm{K}\vdash e_2 : \mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}\mathit{string}\mathrm{K}\mathrm{K}\end{array}}{\mathrm{P}^{-1}\,\mathrm{J}S'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}D'\mathrm{K};\mathrm{P}^{-1}\,\mathrm{J}\Gamma'\mathrm{K}\vdash e_1\ \mathtt{like}\ e_2 : \mathrm{P}^{-1}\,\mathrm{J}\mathrm{P}\,\mathrm{J}\mathit{bool}\mathrm{K}\mathrm{K}}$$

Thus, the desired result is obtained.

4.7 Variable type inference

As discussed in Section 4.5.3 and used in Fig. 4.13, a phantom transformation function for each QQL variable is required for inclusion in the type avatar as the result of the infer_pf. In this section, we discuss how we obtain these functions.

QQL variables can appear in the QQL query clauses; `from`, `where`, `having`, `limit` and `offset`. Variables are not allowed in the `select`, `group-by`, and `order-by` clauses.

Variables in QQL are incorporated in a way consistent with the ML language, i.e. variables are implicitly typed and the type information is inferred during compilation. However, because of using phantom types in the avatar function, we must identify the correct phantom transformation function for each variable during the avatar generation, that is, during pre-processing and *before* compilation.

Variables in the `from` clause can only be of *query-object* type, i.e. they will be bound to queries that can be used as sub-queries in the current context, how their type is obtained and integrated inside the avatar function is discussed in (section 4.8).

During avatar generation (preprocessing time), we do not know what the types of the orm classes and their fields are, they will be inferred by the OCaml compiler in the usual way. In particular, their declarations and mapping information may exist in another source file of the program that has not yet been pre-processed or compiled. Thus if the user writes the query `from customer as c where c.x > :v`, then `v` is a variable, and there is no phantom transformation function for it pre-defined. However, we need such a transformation function so that we can use it in the avatar code we are generating. Even though we do not know the type of `customer.x` at this time, we can tell, if we use type inference, that `v` must have the same type as `customer.x`. Thus, for the query to be well formed, there must be an orm class `customer`, which must have a field called `x`, and the type of this field be the same as the type of variable `v`. Hence, although we do not have sufficient information to resolve the type of `v` fully, we can deduce enough to know that the phantom transformation function that is, or will be, defined for `customer.x` in the `Session` module, will provide the correct transformation for `v` as well. Therefore we use that function in the avatar, and if it turns out that there is no `customer` orm class, or if it does not have a field called `x`, or if the variable `v` gets bound to an incompatible value, then a compilation error will occur when OCaml attempts to compile the generated avatar.

In order to achieve this, we need to infer, not the full types of variables in QQL queries, though we will do so when we can, but only a set of type equations that equate the types of the variables to the types of objects for whom there must be phantom functions in the `Session` module. To do this, we use a variant of a Hindley-Milner (HM) type inference algorithm [Mil78, DM82]. Instead of inferring concrete types (e.g. `int`, `string`), a set of type equations for each variable is identified, where each equation type relates the type of the variable to a standard OCaml type or to the type of an ORM class field. Since the phantom transformation functions for ORM classes and their encapsulated fields are generated according to the class definition (using the transformation rules in section 4.3.2) and implemented inside the ORM session module, we can then obtain the phantom transformation functions for variables based on these type equation.

4.7.1 Type equation set

Since type inference on QQL variables must be performed during the preprocessing phase, there is not enough type information available yet to complete it in this phase as we rely on the OCaml compiler's type inference engine to resolve types between the query and the rest of the OCaml program. Therefore, we construct the type equations on query variables based on primitive types, such as int, float, boolean, etc, and the type of ORM class access and aggregate expressions, the real type of which, since they are unknown at this point, are given as a free type variable. For instance, from the expression `a#b > v`, the type equation τ_v = T(`a#b`) could be inferred, which treats T(`a#b`) as a base type. We connect such types into the OCaml type system at a later stage.

Section 4.2.2 presented the type inference rules of QQL expressions. We would like to use these rules to infer types for variables. However, only for some cases will we be able to fully resolve the type of a variable. For others we can only say that it is the same type as that of the field indicated by a field access expression (e.g. `customer#email}`. There are other possibilities too. We list all of them here:

- τ, where τ is a primitive type (e.g. *bool*, *int*, *string*). An example of an expression where such a type could be resolved for variable v would be `from user where :v > 100`.
- T(ae), the type of a field access expression ae, e.g. `from user where :v = user#email`.

- τ *option*, when the variable is an option-type value. An example where this would occur is `from user where :v is Null`, for checking an option-type variable is `Null`.
- τ *collection*, when the variable is used as a collection, e.g. `from user where user#name in :v`
- $\mathrm{N}_o(\tau)$, represents the internal type of an option type, `from user where (valof user#age) > :v` Clearly, $\mathrm{N}_o(\tau$ `option`$) = \tau$. Also, if $\chi = \tau option$, then $\mathrm{N}_o(\chi)$ $option = \chi$.
- $\mathrm{N}_l(\tau)$, represents the element type in a collection, e.g. `from user where :v in [10; 20; 30]` Clearly, $\mathrm{N}_l(\tau$ $collection) = t$ and, if $\chi = \tau$ $collection$, then $\mathrm{N}_l(\chi)$ $collection = \chi$.

Therefore we need a different set of type inference rules that are based on the proper QQL types but allow us to infer, not full types for variables, but type equations for variables within the constraints imposed by executing this process at pre-processing time. We will then generate code that encapsulates the typing constraints identified in the type equations found so that, at compile time, the OCaml compiler can check these constraints.

4.7.2 Type inference based on type equations

Our inference algorithm is based on, but modified from, the Hindley-Milner (HM) algorithm. It differs from HM in that we construct a set of type equations, which only involves query variables, instead of general type constraints that include all values and expressions in the query. In addition to the construction and unification of type equations, we also identify polymorphic variable types in the query.

Our approach has the following steps: for the query, considering only the combination of `where`, `having`, `offset` and `limit` clauses,

1. Use the inference rules to construct the type equation sets that relate the types of variables to OCaml types, to types of ORM field access expressions or types of other variables.
2. Unify the types in the equations to discover a single, a most informative type for each variable, where the most informative type is a fully resolved OCaml type, the next most informative type is one based on an ORM field expression type, and the least informative is the type of another variable.
3. Generate the phantom functions for each variable from the types found.

At this point we will have phantom functions for every non-polymorphic, non-query type variable in the query, and, as a side effect, will have identified all polymorphic variables. Obtaining the phantom functions for the query variables, will be discussed in Section 4.8.

As a running example, we will consider the following query, which defines three variables in the `where` clause.

```
select e, e#dept#manager
from employee as e
where (e#city in :v1) and
       (e#age between :v2 and :v3)
```

We now discuss the steps of the above approach in more detail in the following subsections.

Step 1: Discovering the type equation sets

The query, at this point, has been parsed into an abstract syntax tree. All our processing on the query from here on works on the AST rather than raw strings.

The fact that we can, in general, only resolve types up to equivalence with types of expressions whose true type will only be discovered later, means that we cannot employ a traditional type inference algorithm. Instead, we introduce a bottom up `type constraint` inference algorithm that follows the shape of the abstract syntax tree to discover the type constraints on variable.

We follow [HHS02] in abandoning the use of a type environment Γ for our type constraint inference. Instead we use an *assumption set*, to record the type variables, rather than types, that are assigned to free variables, and constraint set, to record the type constraints.

A type constraint, $\tau_1 = \tau_2$, indicates that τ_1 and τ_2 should be unified, although it may not be possible to do so immediately.

Our type constraint inference rules, c.f. Fig. 4.17 and 4.18, involves judgements of the form $\mathrm{A}; \mathrm{E} \vdash \mathtt{t} : \tau$ The assumption set A records type variables that are assigned to free variables of `t`. The constraint set, E, is a set of type constraints of the form $\tau_1 = \tau_2$, indicating that τ_1 and τ_2 should be unified at a later stage. This will allow us to infer types up to type equivalence with expressions whose type will only become available after pre-processor time during OCaml compilation. The `t` is the term being typed and the τ is the type entailed for that term under the assumption and constraint sets.

Inference rule VAR introduces a free type variable α, unless . The fact that variable x has this type variable is recorded in the assumption set; the equation set is empty. We assume a scheme of LIT rules for all literals of primitive types, of the form:

$$\frac{}{\emptyset; \emptyset \vdash \mathtt{0} : \mathit{int}}\ \textsc{Lit}_0 \qquad \frac{}{\emptyset; \emptyset \vdash \mathtt{1} : \mathit{int}}\ \textsc{Lit}_1 \qquad \cdots$$

$$\frac{}{\emptyset; \emptyset \vdash \mathtt{`abc'} : \mathit{string}}\ \textsc{Lit}_{\mathtt{`ABC'}} \qquad \cdots$$

$$\vdots$$

ORM field expressions are already identified in the AST, as are all other identifiers, so the type constraint inference rules can use the identification of ORM field access expressions as a side condition. In effect, associating the type of an ORM field access expression with the expression itself is exactly like associating the type of a literal with the literal expression. The ORM field access expressions in our running example query, and their associated types, are:

`e` : *T(*`e`*)*
`e#dept#manager` : *T(*`e#dept#manager`*)*
`e#age` : *T(*`e#age`*)*
`e#city` : *T(*`e#city`*)*

Unlike many type systems, we do not need a function abstraction rule, as there are no function definitions possible in QQL, and we do not need a function application rule, as, although function applications are possible in QQL, only the predefined functions can be applied, for which we have concrete rules in our system.
Then a type derivation tree of the query can be built as in Fig. 4.19, from which assumption set (A) and type constraint set (E) are obtained.

Step 2: Unify the types to find ground types for variables

After the generation of the assumption and the type constraint sets, a series of substitutions is performed to unify the types. This tries to determine an equivalent type expression, without type variables, for each variable. This step is similar to Hindley-Milner solving of type constraints [Pie02], except that substitution in our approach is much simpler as we do not need to consider types like composition types (e.g. *record*, *tuple*) and function types ($\tau \rightarrow \tau'$), nor more complicated sub-typing and recursion. It benefits from the fact that expressions for field and record access are treated essentially as base types, as it is safe to ignore the internal type representation.

Note that the assumption set contains entries of the form $v : \tau$, where v is a QQL variable name and τ is a type variable. There may be multiple entries, with different type variables, for the same QQL variable if that QQL variable occurs more than once in the query. In QQL, all these occurrences refer to the same identifier, a fact that is not true in many programming languages, where creation of an identifier in one scope can "hide" the presence of a different identifier in a different scope. Hence we can simplify our procedure by adding extra type constraints to the constraint set that identifies the type variables associated with the same QQL variables in the assumption set and removing all except one of the type associations for each identifier from the assumption set.

$$\frac{}{\{\mathrm{x}:\alpha\};\emptyset \vdash \texttt{:x} : \alpha}\ \textsc{Var} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{}{\emptyset;\emptyset \vdash \texttt{x} : \textit{T(x)}}\ \text{FAE} \quad \textit{if } \texttt{x} \textit{ is an ORM field access expression}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2}{\mathrm{A}_1 \cup \mathrm{A}_2;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \{\tau_1 = \tau_2\} \vdash e_1\ \texttt{arith}_{\texttt{op}}\ e_2 : \tau_2}\ \textit{Arith}_{OP}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2}{\mathrm{A}_1 \cup \mathrm{A}_2;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \{\tau_1 = \tau_2\} \vdash e_1\ \texttt{compare}_{\texttt{op}}\ e_2 : \textit{bool}}\ \textit{Compare}_{OP}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2}{\mathrm{A}_1 \cup \mathrm{A}_2;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \{\tau_1 = \text{bool};\ \tau_2 = \text{bool}\} \vdash e_1\ \texttt{and}\ e_2 : \textit{bool}}\ \textsc{And}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2}{\mathrm{A}_1 \cup \mathrm{A}_2;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \{\tau_1 = \text{bool};\ \tau_2 = \text{bool}\} \vdash e_1\ \texttt{or}\ e_2 : \textit{bool}}\ \textsc{Or}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \text{bool}\} \vdash \texttt{not}\ e : \textit{bool}}\ \textsc{Not}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\} \vdash \texttt{count}\ e : \textit{int}}\ \textsc{Count} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\} \vdash \texttt{sum}\ e : \alpha}\ \textsc{Sum} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\} \vdash \texttt{avg}\ e : \alpha}\ \textsc{Avg} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\} \vdash \texttt{max}\ e : \alpha}\ \textsc{Max} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\} \vdash \texttt{min}\ e : \alpha}\ \textsc{Min} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \text{collection}\} \vdash \texttt{e\#count()} : \textit{int}}\ \textsc{Count-col} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \text{collection}\} \vdash \texttt{e\#sum()} : \alpha}\ \textsc{Sum-col} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \text{collection}\} \vdash \texttt{e\#avg()} : \alpha}\ \textsc{Avg-col} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \text{collection}\} \vdash \texttt{e\#max()} : \alpha}\ \textsc{Max-col} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \text{collection}\} \vdash \texttt{e\#min()} : \alpha}\ \textsc{Min-col} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \text{bool}\} \vdash \texttt{where e} : \textit{bool}}\ \textsc{Where}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \text{bool}\} \vdash \texttt{having e} : \textit{bool}}\ \textsc{Having}$$

Figure 4.17: Type Inference Rules for Constructing Equation Set

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \mathrm{int}\} \vdash \texttt{limit e} : \mathit{int}}\ \textsc{Limit}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \mathrm{int}\} \vdash \texttt{offset e} : \mathit{int}}\ \textsc{Offset}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \mathrm{option}\} \vdash \texttt{valof e} : \alpha}\ \textsc{Valof} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A};\mathrm{E} \vdash e : \tau}{\mathrm{A};\mathrm{E} \cup \{\tau = \alpha\ \mathrm{collection}\} \vdash \texttt{itemsof e} : \alpha}\ \textsc{Itemsof} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2 \qquad \cdots \qquad \mathrm{A}_\mathrm{n};\mathrm{E}_\mathrm{n} \vdash e_n : \tau_n}{\bigcup_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{A}_\mathrm{i}; \bigcup_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{E}_\mathrm{i} \cup \{\tau_1 = \tau_2, \ldots, \tau_1 = \tau_\mathrm{n}, \alpha = \tau_1\ \mathrm{collection}\} \vdash \texttt{[}e_1\texttt{; } e_2\texttt{; } \cdots\texttt{; } e_n\texttt{]} : \alpha}\ \textsc{List} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2}{\mathrm{A}_1 \cup \mathrm{A}_2;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \{\tau_2 = \tau_1\ \mathrm{collection}\} \vdash e_1\ \texttt{in}\ e_2 : \mathit{bool}}\ \textsc{In}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2}{\mathrm{A}_1 \cup \mathrm{A}_2;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \{\tau_1 = \mathrm{string};\ \tau_2 = \mathrm{string}\} \vdash e_1\ \texttt{like}\ e_2 : \mathit{bool}}\ \textsc{Like}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e_1 : \tau_1 \qquad \mathrm{A}_2;\mathrm{E}_2 \vdash e_2 : \tau_2 \qquad \mathrm{A}_3;\mathrm{E}_3 \vdash e_3 : \tau_3}{\mathrm{A}_1 \cup \mathrm{A}_2 \cup \mathrm{A}_3;\mathrm{E}_1 \cup \mathrm{E}_2 \cup \mathrm{E}_3 \cup \{\tau_1 = \tau_2;\ \tau_1 = \tau_3\} \vdash e_1\ \texttt{between}\ e_2\ \texttt{and}\ e_3 : \mathit{bool}}\ \textsc{Between-and}$$

$$\frac{\mathrm{A}_1;\mathrm{E}_1 \vdash e : \tau}{\mathrm{A}_1;\mathrm{E}_1 \cup \{\tau = \alpha\ \mathrm{option}\} \vdash e\ \texttt{is [not]}^{0.1}\ \texttt{Null} : \mathit{bool}}\ \textsc{Null} \quad \textit{where } \alpha \textit{ is a fresh type variable}$$

Figure 4.18: Type Inference Rules for Constructing Equation Set (cont.)

The constraint set will hold equations between different type expressions. The type expressions that are possible are limited by the grammar of QQL. In practice, they can be divided into two types, *determined* or *undetermined*.

Determined types have no type variables in them. They can *only* take one of the following forms:

- *t*, where *t* is a base type, e.g. *int*, *string*, etc.
- *t option*, where *t* is a base type, e.g. *int*, *string*, etc.
- *t collection*, where *t* is a base type, e.g. *int*, *string*, etc.
- *T(e)*, where *e* is an orm field access expression.
- *T(e) collection*, where *e* is an orm field access expression.
- *T(e) option*, where *e* is an orm field access expression.
- *No(e)*, where *e* is an orm field access expression.
- *Nl(e)*, where *e* is an orm field access expression.

Undetermined types have type variables in them. They can **only** take one of the following forms:

- θ, where θ is a type variable. This results from an expression such as "`:v1 > :v2`".

$$\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset \vdash \texttt{e\#city} : \textit{T(e\#city)}} \quad \dfrac{}{\mathrm{A}_1;\emptyset \vdash \texttt{:v1} : \tau_1}}{\mathrm{A}_1;\mathrm{E}_1 \vdash \texttt{e\#city in :v1} : \textit{bool}} \quad \dfrac{\dfrac{}{\emptyset;\emptyset \vdash \texttt{e\#age} : \textit{T(e\#age)}} \quad \dfrac{}{\mathrm{A}_2;\emptyset \vdash \texttt{:v2} : \tau_2} \quad \dfrac{}{\mathrm{A}_3;\emptyset \vdash \texttt{:v3} : \tau_3}}{\mathrm{A}_2 \cup \mathrm{A}_3;\mathrm{E}_2 \vdash \texttt{(e\#age between :v2 and :v3)} : \textit{bool}}}{\mathrm{A}_1 \cup \mathrm{A}_2 \cup \mathrm{A}_3;\mathrm{E}_1 \cup \mathrm{E}_2 \vdash \texttt{(e\#city in :v1) and (e\#age between :v2 and :v3)} : \textit{bool}}$$

$$\mathrm{A}_4;\mathrm{E}_3 \vdash \texttt{where (e\#city in :v1) and (e\#age between :v2 and :v3)} : \textit{bool}$$

$$\mathrm{A}_1 = \{\texttt{v1} : \tau_1\} \qquad \mathrm{A}_2 = \{\texttt{v2} : \tau_2\} \qquad \mathrm{A}_3 = \{\texttt{v3} : \tau_3\} \qquad \mathrm{A}_4 = \{\texttt{v1} : \tau_1, \texttt{v2} : \tau_2, \texttt{v3} : \tau_3\}$$

$$\mathrm{E}_1 = \{\tau_1 = \textit{T(e\#city) collection}\} \qquad \mathrm{E}_2 = \{\tau_2 = \textit{T(e\#age)}, \tau_3 = \tau_2\}$$

$$\mathrm{E}_3 = \{\tau_1 = \textit{T(e\#city) collection}, \tau_2 = \textit{T(e\#age)}, \tau_3 = \tau_2\}$$

Figure 4.19: Type Derivation Tree

- θ *collection*, where θ is a type variable. This results from an expression such as "`:v1 in :v2`".
- θ *option*, where θ is a type variable. This results from an expression such as "`:v1 is null`".

Note that there are no more complicated forms involving deeper nesting of type variables.

With these definitions of determined and undetermined type expressions, we can now define determined and undetermined type equations. A determined type equation is an equation of type expressions where one of the sides of the equation is a determined type expression and the other side is either a type variable or a determined type expression. All other type equations are undetermined.

Before presenting our type unification algorithm, we first introduce an ancillary function which, by rewriting constraint equations into a standard, simpler form, simplifies the main algorithm.

Algorithm 1 (Simplify)

INPUT:

A constraint equation $x = y$.

OUTPUT:

A equivalent constraint equation $v = w$, in a standard, simpler form, i.e. if one of x and y contains a type variable and the other does not, then v will be a type variable and w will contain no type variables where $v = w$ if and only if $x = y$. Otherwise it returns $x = y$ unchanged.

METHOD:

1. If neither or both of x and y contain a type variable, then return $x = y$.
2. If y contains a type variable, let (s, t) be (y, x), else let (s, t) be (x, y).
3. If s is of the form τ, where τ is a type variable, then return $s = t$.
4. If s is of the form τ *collection*, where τ is a type variable, then return $\tau = \textit{Nl(t)}$.
5. If s is of the form τ *option*, where τ is a type variable, then return $\tau = \textit{No(t)}$.

The main unification algorithm proceeds as follows:

Algorithm 2 (Unify)

INPUT:

An assumption set A of variable name, type variable pairs and a set E of type constraint equations

OUTPUT:

A triple (R, U, D),

- where $R \subseteq A$ such that for every entry in A, there is an entry in R with for the same QQL variable, but no pair of elements of R is has the same QQL variable but different type variable associations, and
- where U is a set of undetermined type equations and D is a set of determined type equations such that $U \cup D$ is the unification of E and all entries in D are in simplified form, i.e. $\forall d \in D.\ \mathit{Simplify}(d) = d$.

METHOD:

1. Create empty *determined*, D, and *undetermined*, U, constraint sets.
2. For each constraint, $x = y$, in the original constraint set, add $\mathit{Simplify}(x = y)$ to D if either or both type expressions x or y does not contain a type variable. Otherwise add $x = y$ to U.
3. Let R be a copy of A.
4. While there remains a pair of entries $v : \tau_1$, $v : \tau_2$ in R, for the same QQL variable, v, eliminate one of them from R and add $\tau_1 = \tau_2$ to U.
5. At this point, the only constraints remaining in U are of the form $\tau_1 = \tau_2$, $\tau_1 = \tau_2$ *option*, τ_2 *option* $= \tau_1$, $\tau_1 = \tau_2$ *collection*, or τ_2 *collection* $= \tau_1$, where $\tau_1 \neq \tau_2$. Further, any type constraints in D that contain a type variable τ are of the form $\tau = e$, where e does not contain any type variables.

   While there is a type constraint in D of the form $\tau = e$, and a type constraint u in U involving τ, where τ is a type variable;

   (a) If u is of the form $\tau = \tau_1$ or $\tau_1 = \tau$, then remove u from U and add $\tau_1 = e$ to D.

   (b) If u is of the form $\tau = \tau_1$ *option* or τ_1 *option* $= \tau$, then remove u from U and add $\tau_1 =$ *No(e)* to D.

   (c) If u is of the form $\tau_1 = \tau$ *option* or τ *option* $= \tau_1$, then remove u from U and add $\tau_1 = e$ *option* to D.

   (d) If u is of the form $\tau = \tau_1$ *collection* or τ_1 *collection* $= \tau$, then remove u from U and add $\tau_1 =$ *Nl(e)* to D.

   (e) If u is of the form $\tau_1 = \tau$ *collection* or τ *collection* $= \tau_1$, then remove u from U and add $\tau_1 = e$ *collection* to D.
6. return (R, U, D)

To clarify the process we show how to unify the example set E_3 generated in the previous step. For this example, there are no pairs of entries in the assumption set with the same QQL variable, so no such pairs need to be reduced. Thus E_3 is divided into sets U and D, where $U = \{\tau_3 = \tau_2\}$, and $D = \{\tau_1 = T(\texttt{e\#city})\ \mathit{collection}, \tau_2 = T(\texttt{e\#age})\}$. Here we can now apply one substitution as described in step 5 above, to get

$$U = \emptyset$$
$$D = \{\tau_1 = T(\texttt{e\#city})\ \mathit{collection}, \tau_2 = T(\texttt{e\#age}), \tau_3 = T(\texttt{e\#age})\}$$

We get a fully determined equation set in this example.

Step 3: Generate phantom functions for each variable

At this point, we have an assumption set A, containing the mapping of QQL variable names to type variables, the determined set D of determined type equations that have been unified from the original set of type equations, and the undetermined set U of remaining undetermined (and undeterminable) type equations.

For each QQL variable, v, we obtain its corresponding determined type equation, if it exists, by finding the type variable associated with the QQL variable, θ from A, and then finding any type equation in D between that type variable and a determined type expression τ. We then generate a

phantom transformation function for v to include in the type avatar as the result of the `infer_pf` as discussed in Section 4.5.3 and used in Fig. 4.13. If the corresponding determined type equation does not exist, then the QQL variable involved is polymorphic. This case is discussed below.

As discussed in section 4.7.1, the determined equivalent type consists of primitive types, types of field access expressions, or a small number of functions of these types. The primitive types and built-in functions have corresponding transformation functions predefined in the `Phantom` module. The `Session` module, which is generated during pre-processing, contains the necessary transformation functions for field access expressions. To turn a field access expression into the appropriate transformation function, we need to analyse the expression to obtain the full name of the required function. The function *popf* carries out this analysis and is based on a careful convention for how these transformation functions are structured in the Phantom module. This will be discussed in more detail in section 4.8, but, briefly, for a field access expression of the form `project#manager#dept#name`, the appropriate transformation function will be `Session.project#manager#dept#name#phantom`, i.e. the `Session` module will have an object for every orm class and field. These objects are not of the orm classes, but match the structure of those classes so that the same field traversals are possible in these objects as in the objects of the corresponding orm classes. For every field, there is a method `phantom`, that returns the phantom function for that field. Thus *popf* merely has to recurse over the structure of the field access expression, generating corresponding `Session` object graph traversal calls. It uses a recursive function `pobj` to return the phantom transformation object expression (which contains method `phantom` for the phantom transformation) for the given field expression. The function `convexpr` is used only in map style field access expressions. It converts various forms of such map style field access expressions to object field access expressions, and removes the local variable `id`:

popf(*x*) = *pobj*(*x*)`#phantom`

pobj(*id*) = `Session.`*id* where *id* is an orm class; *id*`#phantom_transformation_context` where *id* is of query type

pobj(*ae*`#`*id*) = *pobj*(*ae*)`#`*id*

pobj(*ae*`.`*id*) = *pobj*(*ae*)`#`*id*

pobj(`valof` *ae*) = *pobj*(*ae*)`#valof`

pobj(`itemsof` *ae*) = *pobj*(*ae*)`#itemsof`

pobj(*ae*`#[`*id* `=>` ae_2`]`) =

```
object
  val pf = pobj(ae)#itemsofconvexpr(ae2, id)
  method phantom =
    fun v -> map (fun x -> pf#phantom x) v
  method itemsof = pf
end
```

convexpr(*id*, *id*) =

convexpr(*ae*`#`*f*, *id*) = *convexpr*(*ae*, *id*)`#`*f*

convexpr(*ae*`.`*f*, *id*) = *convexpr*(*ae*, *id*)`#`*f*

convexpr(`valof` *ae*, *id*) = *convexpr*(*ae*, *id*)`#valof`

convexpr(`itemsof` *ae*, *id*) = *convexpr*(*ae*, *id*)`#itemsof`

convexpr(*ae*`#[`*x* `=>` ae_2`]`, *id*) = *convexpr*(*ae*, *id*)`#itemsof`*convexpr*(ae_2, *x*)

The value of `infer_pf`(v) will depend on the type expression τ. We evaluate it by a case analysis on the form of τ:

1. If τ is of primitive type, e.g. *int*, *float*, *string*, then

   `infer_pf`$(v) =$ *pvt*(τ)

2. If τ is of the form: `T`(e), then

   `infer_pf`$(v) =$ *popf*(e)

3. If τ is of the form: τ' *option* and τ' is of primitive type, then

```
infer_pf(v) = fun v -> Phantom.univalued(
                  Phantom.optionize (match v with
                                     | Some x -> Some (put(τ') x)
                                     | None -> None))
```

4. If τ is of the form: τ' *option* and τ' is of the form: T(e), then

```
infer_pf(v) = fun v -> Phantom.univalued(
                  Phantom.optionize (match v with
                                     | Some x -> Some (popf(e) x)
                                     | None -> None))
```

5. If τ is of the form: τ' *collection* and τ' is of primitive type, then

```
infer_pf(v) = fun v -> map (fun x -> put(τ') x) v
```

6. If τ is of the form: τ' *collection* and τ' is of the form: T(e), then

```
infer_pf(v) = fun v -> map (fun x -> popf(e) x) v
```

7. If τ is of the form N_o(T(e)), then

```
infer_pf(v) = fun v -> Library.valof (Phantom.get (popf(e) (Some v)))
```

8. If τ is of the form N_l(T(e)), then

```
infer_pf(v) = fun v -> Collection.hd (popf(e) new collection [v]))
```

Polymorphic QQL variables: The unification determines equivalent types for most query variables, however, there may still remain undetermined variables, which are of polymorphic types and caused by binary operations like comparison or between-and operations between QQL variables which are not otherwise used in operations or related to ORM field access expressions in the query. An example would be:

```
from employee as e
where :v1 = :v2 and
      :v3 in :v4 and
      :v5 is not null and
      :v6 between :v7 and :v8
```

In these cases, although the where condition will evaluate constantly to true or false for all tuples in `employee` depending only on the values passed into the query, it is still a valid query according to QQL syntax and semantics. All we can deduce about the types is that the types of `v1` and `v2` must be the same, the type of `v4` is a collection of the type of `v3`, that `v5` is an option type of some unknown type and that `v6`, `v7`, and `v8`, are of the same, comparable, type.

We have identified such variables simply by being unable to resolve their type to a determined type. The avatar generation procedure, c.f. section 4.5.3, uses this information to generate code that enforces the same polymorphic constraints in the avatar, without restrict the specific type of the variables involved. Thus our example above would have the following code generated:

```
:v1 = :v2 ⇝ Phantom.mkBool (v1 = v2)
:v3 in :v4 ⇝ Phantom.mkBool (Collection.mem v3 v4)
:v5 is not null ⇝ Phantom.mkBool (Library.isSome v5)
:v6 between :v7 and :v8 ⇝ Phantom.mkBool (Library.between v6 (v7 v8))
```

The corresponding code generated for the where clause, therefore, would be:

```
PhantomFunctions.where (
   PhantomFunctions.and
      (PhantomFunctions.and
         (PhantomFunctions.and (Phantom.mkBool (v1 = v2))
              (Phantom.mkBool (Collection.mem v3 v4)))
         (Phantom.mkBool (Library.isSome v5)))
      (Phantom.mkBool (Library.between v6 (v7 v8))))
```

Note that the four situations described in this example are the only situations in which polymorphic variable constraints can occur. This fact is used in the avatar generation to capture precisely the expressions that need to be dealt with to handle these constraints.

4.8 Support for query composition

Constructing complex queries out of smaller pieces is commonly achieved in many programming language interfaces to databases by means of type unsafe SQL string concatenation. One of the advantages of QQL over such frameworks is its support for type-safe query composition by incorporating queries as variables in another query. For example, Consider the following queries, where, in each case, we show the creation of the query in the first statement and the execution of the query in the second:

1. Find employees who have worked on more than `np` projects.

   ```
   let q_emp = Session.[from employee as emp where emp#projects#count() > :np]

   Session.query sess (q_emp ~np:10)
   ```

2. Find projects whose funding is over `nf` pounds.

   ```
   let q_proj = Session.[from project as proj where proj#fund > :nf]

   Session.query sess (q_proj ~nf:(Numeric.of_string "250000"))
   ```

3. Find employee project pairs, where the employee has worked on more than `np` projects, the project has funding of over `nf` pounds and the employee is the manager of this project.

   ```
   let q3 = Session.[select q1#emp as emp, select q2#proj as proj
                     from :q_employee as q1 join :q_project as q2
                          on q1#emp#id = q2#proj#manager#id]

   Session.query sess (q3 ~q_employee:(q_emp ~np:20)
                          ~q_project:(q_proj ~nf:(Numeric.of_string "250000")))
   ```

In order to achieve this kind of compositionality, QQL queries are compiled (translated) into self-contained objects, or functions to generate this object when variables are defined in the query. By 'self-contained', we mean that the compiled query object contains all necessary data, such as a query expression tree for translation to SQL, avatar data for compile-time type safety, a phantom transformation context used in the query composition for providing phantom transformation functions for the query return values, and a result construction object for constructing objects from the return value.

We have discussed type avatars in section 4.5, and will discuss how to perform the translation from a QQL query into SQL in section 4.9. Libraries like SQLJ [Ame99] translate queries into SQL during pre-processing. However, because of the support for query composition, QQL defers this translation until runtime when the query is about to execute.

The query phantom-transformation context (*pt-context*) acts as a repository of phantom transformation functions when this query is used in a query composition. Specifically, this context is designed as an object structure consisting of a method for transforming the query return value into its corresponding phantom-type value and methods for performing the phantom transformation of each query `select` element.

This pt-context object must contain transformation functions for each individual returned *select* field, as well as for the query return value, since the phantom transformation function of these fields may be used to convert a type-equivalent variable in the avatar function. However, it becomes difficult to record these phantom transformation functions in the context object when the query is built based on a sub-query (i.e. using query composition) and parts of the *select* fields are constructed from a sub-query that is unknown at the time the current query is defined. This requires a unified approach to find and record transformation functions for queryable orm classes and sub-queries, since both of them can be used in the `from` clause. Further, we require that the approach makes it possible to pt-context object during preprocessing time without knowing the concrete sub-query that is going to be passed to this query.

Recall in the object graph, objects holds references to others, then programmers can navigate following these references. This provides the possibility to wrap phantom transformation functions for orm classes and their fields as objects, and organise these objects in an object graph that exactly reflects the structure of orm classes and their associations. This implies that we can get the phantom transformation function for one associated field of the current entity by navigating the current entity's transformation-function object. For example. to find the phantom transformation function for the field `age` of orm class `User`, we get the `phantom` method function of the `age` object of the `User` object in the `Session` module. Thus we achieve the effect of getting transformation functions in a navigational way. Such an object graph for orm classes is pre-generated in the ORM session module for global use; thus, the query pt-context object only records references of these transformation functions, and wraps and organise these transformation functions as objects in a way similar to the one for orm classes. This allows locating phantom transformation functions of query result navigationally in the case the query is used as a sub query in the query composition.

Figure 4.20 defines the generation of phantom transformation objects from orm class definitions. $\mathrm{G}\,\mathrm{J}x\mathrm{K}$ defines the generation of phantom transformation objects for type x; $pvt(\tau)$ denotes the phantom transformation function for type τ (see Figure 4.7). Each phantom transformation object contains a method, named `phantom`, which is the real transformation function for the value, and methods for returning the transformation object of the value's encapsulated fields. Note that when a field is of type `option` or `collection`, the corresponding transformation object includes additional methods named as `valof` or `itemsof` for the transformation object of values extracted from this option type, or elements in this collection respectively. When a field is of an associated orm type, its transformation object is obtained by initialising the corresponding orm class's transformation object. The phantom transformation objects for orm classes are pre-generated in the Session module for global use and assigned the same name as the name of the orm class.

For concreteness, we show an example generation of phantom transformation objects. In this example, orm class `employee` contains an option-type integer, `id` and a collection of related `projects`. Orm class project contains a string-type name and numeric-type (Qanat's numerical type used for currency values) `fund`, and an associated field `manager` of type `employee`:

```
(* we omit the meta key and bind definition for conciseness *)
class orm employee : "employees"= object
  val mutable id : int {auto}
  val mutable projects : project resizeSet with key_emp_project on save_update
end
and project : "projects" = object
  val mutable name : string {assign}
  val mutable fund : Numeric.t
  val manager : employee with key_manager on save_update
end
```

Figure 4.21 presents the pre-generated phantom transformation objects for orm class `employee` and `project`. Since `employee` and `project` are defined as recursive classes, the classes for phantom transformation objects are also generated in a recursive way. The `Session` module includes global references to the phantom transformation objects of `employee` and `project`. Thus the phantom transformation functions for `project` and `project#manager` can be obtained using expression `Session.project#phantom` and `Session.project#manager#phantom`, respectively.

Based on these pre-generated objects, phantom transformation functions for query return values and for each *select* element are generated and organised in an object structure that mirrors that of

```
(* orm class definition *)
class orm x = object
  (* orm class fields *)
  val mutable *f0 : t0 {key-gen}
      ⋮
  val mutable fn : tn = vn

  (* user defined methods *)
  method uf_m = ··· (* function impl *)
end
```

G J ⋮ K =

```
(* phantom transformation class and object *)
class pto_x = object
  (* transformation function for class x *)
  method phantom = pvt(x)
  (* transformation object for field fi *)
  method f_i = G J t_i K
  (* no user defined methods in pto_x *)
end

let x = new pto_x
```

```
(* The phantom transformation object for class x is initialized by using expression:
        new pto_x *)

(* primitive type *)
G Jt K = object method phantom = pvt(t) end

(* option type *)
G Jτ option K = object
                    method phantom = pvt(τ option)
                    method valof = G Jτ K
                end
(* tuple type *)
G Jτ1 × ··· × τn K = object method phantom = pvt(τ1 × ··· × τn) end

(* record type *)
G J{r1 : τ1; ...; rn : τn} K = object
                                  method phantom = pvt({r1 : τ1; ...; rn : τn})
                                  method r1 = G Jτ1 K
                                  ...
                                  method rn = G Jτn K
                               end
(* variant type *)
G J C1 ⟨of τ11, ..., τ1j⟩ |...| Cn ⟨of τn1, ..., τnk⟩ K =
          object
             method phantom = pvt(C1 ⟨of τ11, ..., τ1j⟩ |...| Cn ⟨of τn1, ..., τnk⟩)
          end

(* anonymous object *)
G J<m1 : τ1; ...; mn : τn> K = object
                                   method phantom = pvt(<m1 : τ1; ...; mn : τn>)
                                   method m1 = G Jτ1 K ... method mn = G Jτn K
                               end
(* orm class *)
G Jx K = new pto_x

(* orm-class collection *)
G Jx collection K = object
                        (* phantom transformation object *)
                        val pobj = new pto_x
                        method phantom = fun vs -> map(fun x -> pobj#phantom x) vs
                        method itemsof = pobj
                    end
```

Figure 4.20: Generation of Phantom transformation objects

```
(* Session module with pre-generated phantom transformation objects *)
module Session = struct
  (* phantom transformation functions for class employee and project *)
  let rec pf_employee emp =
     Phantom.queryable(Phantom.univalued (Phantom.init (
          object
            method id = Phantom.univalued (Phantom.optionize (
                        match emp#id with
                        |Some i -> Some (Phantom.mkInt i)
                        |None -> None))
            method projects = map(fun v -> pf_project v) emp#projects
          end)))
  and pf_project proj =
     Phantom.queryable(Phantom.univalued (Phantom.init (
          object
            method name = Phantom.mkString proj#name
            method fund = Phantom.mkNumeric proj#fund
            method manager = pf_employee proj#manager
          end)))


  (* phantom-transformation-object class for class employee and project *)
  class pto_employee = object
     (* method for returning phantom transformation function for employee *)
     method phantom = fun emp -> pf_employee emp

     (* method for returning phantom transformation object for each field *)
     method id = object
                  method phantom = fun id ->
                    Phantom.univalued (Phantom.optionize (
                      match id with
                      |Some i -> Some (Phantom.mkInt i)
                      |None -> None))

                  method valof = object method phantom = Phantom.mkInt end
                end
     method projects = object
                  val pobj = new pto_project
                  method phantom = fun vs -> map(fun x -> pobj#phantom x) vs
                  method itemsof = pobj
                end
  end
  and pto_project = object
     method phantom = fun proj -> pf_project proj
     method name = object method phantom = Phantom.mkString end
     method fund = object method phantom = Phantom.mkNumeric end
     method manager = new pto_employee
  end

  (* global phantom-transformation-object for class employee and project *)
  let employee = new pto_employee
  let project = new pto_project
end
```

Figure 4.21: Pre-generated phantom transformation object for orm classes

the orm classes. Thus the query pt-context object contains a `phantom` method for the translation of the query result, and methods for obtaining the phantom translation object of individual *select* fields.

For example, the pt-context object for the query demonstrated in the beginning of this section can be generated as shown in Figure 4.22. Consider *query 3*, which returns `emp` and `proj` by default. If this query is used, in turn, as a sub-query during another query composition, the phantom transformation function for field `id` of `manager` of `proj` in this query result, i.e. `:q#proj#manager#id`, can be obtained using expression `q#phantom_transformation_context#proj#manager#id#phantom` in our usual navigational way.

In a similar way, each query object is compiled with a result construction object for constructing objects from the tabular results. The similarity means that global result construction objects for orm objects and fields are pre-generated in the session module and organised in line with the associations between orm classes and fields. The result construction object for a query contains references to these pre-generated construction objects, thus providing access to the construction function for any returned field in a navigational way.

```
query 1:
  Session.[from employee as emp where emp#projects#count() > :np]

  let phantom_transformation_context =
      object(self)
        (* locate phantom transformation object for orm class used in the query.
           Note that transformation objects for orm classes are pre-generated in
           Session module, and assigned a name the same as the one of the orm
           class *)
        val pto_emp = Session.employee

        method emp = pto_emp

        (* phantom transformation function for query result. Note that, because
           query can be used as sub-queries in query composition (i.e. the query
           result can act as a queryable source), it is necessary to set the
           "queryable" tag for the query return value *)
        method phantom = fun v ->
          let emp_v = v in
          Phantom.queryable(Phantom.init(
            object method emp = self#emp#phantom emp_v end))
      end
```

```
query 2:
  Session.[from project as proj where proj#fund > :nf]

  (* similar to the above query *)
  let phantom_transformation_context =
      object(self)
        (* locate phantom transformation object for orm class used in the query. *)
        val pto_proj = Session.project

        method proj = pto_project

        method phantom = fun v ->
          let proj_v = v in
          Phantom.queryable(Phantom.init(
            object method method proj = self#emp#phantom proj_v end))
      end
```

```
query 3:
  Session.[select q1#emp as emp, q2#proj as proj
           from :q_employee as q1 join :q_project as q2
               on q1#emp#id = q2#proj#manager#id]

  let phantom_transformation_context =
      object(self)
         (* locate phantom transformation object (i.e. the query context object)
            for subqueries *)
        val pto_emp = q_employee#phantom_transformation_context
        val pto_proj= q_project#phantom_transformation_context

        (* methods for returning phantom transformation object for each returned
           element *)
        method emp = pto_emp#emp
        method proj = pto_proj#proj

        method phantom = fun v ->
          let (emp_v, proj_v) = v in
          Phantom.queryable(Phantom.init
             (object
                method emp = self#emp#phantom emp_v
                method proj = self#ename#phantom proj_v
              end))
      end
```

Figure 4.22: Query Phantom Transformation Context Example

4.9 Translation from QQL to SQL

To execute a QQL query against a relational database, it has to be translated into SQL. This section deals with the translation of QQL into equivalent SQL. By providing a translation T, we show how a QQL query Q is translated into an SQL query S, i.e. $S = T(Q)$.

There have been a number of papers investigating query translation from an object-oriented query to the relational query, [KKM93, UA94, YZM+95, HK03]. Much of the effort already spent in this field can be adapted. A general translation approach [KKM94, GS96, Gru99, Jin99, FM00, Gru03] employs monoid comprehension as the intermediate representation of query constructs. By understanding bulk types as a monoid algebraic structure, different collection types can be treated in a uniform manner [Feg94]. Comprehensions [JW07] provide a convenient notation, close to relational calculus, for computations carried out on monoid elements. With this approach it becomes natural to map object-oriented queries to a monoid comprehension calculus, which can easily be translated to equivalent SQL. We adapt the same approach for the translation of QQL.

4.9.1 Monoid comprehension

The monoid calculus [Feg94] allows treating various bulk types in a uniform manner, based on the fact that bulk data (e.g. sets, bags and lists) can be modelled as monoids. We review here the variation of the monoid calculus described in [GS96].

A monoid $\mathcal{M}$ is an algebraic structure consisting of a set with two basic properties:

1. an associative binary operation, $\mathsf{merge}[\mathcal{M}]$.
2. an identity element, $\mathsf{zero}[\mathcal{M}]$, representing the basic monoid unit,

These monoid properties correspond to the usual merge operation and unit element of collection types such as sets, bags and lists. For this reason, given a monoid $\mathcal{M}$, a collection containing elements of type α can be modelled by the type $\alpha\ \mathcal{M}$. We therefore further add a constructor function, $\mathsf{unit}[\mathcal{M}]$, which constructs single element collections from an element. Thus we can describe a collection type by giving the tuple $(\mathsf{zero}[\mathcal{M}], \mathsf{unit}[\mathcal{M}], \mathsf{merge}[\mathcal{M}])$. Within this monoid context, types like set and bag can be represented as the triple $(\{\}, \{\cdot\}, \cup)$ and $(\{\!\{\}\!\}, \{\!\{\cdot\}\!\}, \uplus)$, having the empty collection as the identity, and union as the merge operation. Thus, a set containing elements from 1 to 3 can be constructed as,

$$\begin{aligned}&\mathsf{merge}[\mathsf{set}]\,(\mathsf{unit}[\mathsf{set}](1), \mathsf{merge}[\mathsf{set}](\mathsf{unit}[\mathsf{set}](2), \mathsf{unit}[\mathsf{set}](3)))\\ &\quad = \{1\} \cup (2 \cup 3)\\ &\quad = \{1, 2, 3\}\end{aligned}$$

A monoid is called primitive if it takes an identity function, id, as its unit operation, rather than being free over its base type (which would correspond to representing a collection). For example, we define $\mathsf{sum} = (0, id, +)$ as a primitive monoid, which uses plus as the merge operation. A shorthand $\mathcal{M}\{e_1, \ldots, e_n\}$, for any monoid $\mathcal{M}$, represents the execution of cascading $merge[\mathcal{M}]$ operations over the respective $\mathsf{unit}[\mathcal{M}]e_i$. For instance, an aggregated sum operation on elements 1 to 3 can be defined as,

$$\begin{aligned}&\mathsf{sum}\{1, 2, 3\}\\ &\quad = \mathsf{merge}[\mathsf{sum}]\,(\mathsf{unit}[\mathsf{sum}](1), \mathsf{merge}[\mathsf{sum}](\mathsf{unit}[\mathsf{sum}](2), \mathsf{unit}[\mathsf{sum}](3)))\\ &\quad = 1 + (2 + 3)\\ &\quad = 6\end{aligned}$$

By analogy to list comprehensions [TW89, BLS+94, JW07], expressions of the form $\mathcal{M}\{c|e_1, \ldots, e_n\}$ defines a comprehension over monoid $\mathcal{M}$, the value of which is constructed from elements that satisfy the conditions specified by e_i. Term c defines the constructor of result elements. e_i either generates or filters out the values c gets bound to:

- If e_i is of the form $x \leftarrow E$, it defines a generator, sequentially binding variable x to the elements of E, where E could be another monoid, variable x can be referred to in $e_{i+1} \ldots e_n$ and in c.
- Otherwise, e_i acts as a filter predicate, indicating whether the current generator variable bindings will be propagated further, i.e. if e_i evaluates to false, the constructed c will not be accounted as a result element.

We add to this notation binding expressions of the form $v \equiv e$, which makes the new variable v a synonym for the expression e in all subsequent qualifiers and in the head of the comprehension.

Finally, values constructed during the process of comprehension evaluation are sequentially accumulated via $merge[\mathcal{M}]$. For example, the following comprehension expression results in an integer set containing 2 and 6.

$$\mathsf{list}\{x \times 2 | x \leftarrow \mathsf{list}\{1,2,3\}, x \neq 2\} = [2,6]$$

The generator domain E can itself be a comprehension, leading to nested comprehensions. Rewrite systems for nested comprehensions [Won93, Feg94] can flatten nested comprehensions into multiple generator comprehensions.

4.9.2 From QQL to SQL

This section shows that QQL can be translated into SQL by leveraging monoid comprehensions as an intermediate representation. The mapping of ODMG OQL constructs into monoid comprehensions has already been described in the literature [FM95, GS96]. We adjust this mapping to fit QQL, and transform monoid comprehensions into a form close to relational calculus by resolving navigational expressions and nested return values. We then use this representation to obtain the equivalent SQL.

Let e_i and E_i denote well typed QQL expressions, x_i and k_i denote binding names, E_p and E_f denote boolean predicates, $\bar{x}$ denotes a number of values x_i, $\equiv$ inside a comprehension structure denotes a value binding. We directly use monoid functions (e.g. `sum`, `sorted`) originally defined in [GS96, Gru99].

SQL allows the overloading of table names as tuple variables. Consider the following SQL query:

```
SELECT user.name
FROM user
```

Here `user` is used twice, in the `SELECT` clause as a tuple variable that takes the value of a row from the `user` table, and in the `FROM` clause as the name of the `user` table, i.e. as the collection of rows in the `user` table. Since we want QQL to be an "objectified" version of SQL, we also support this kind of name overloading. However, we have to be careful about such use because, due to our support for navigational expressions, collections can appear in places in QQL queries, such as in `select` clauses, where they could not in SQL. For this reason, we annotate any occurrence of the use of a collection identifier or expression x, in a monoid comprehension, as $\underline{x}$.

1. **select-from-where** By understanding query sources as monoids, `select-from-where` can be represented as a form of multiple-generator comprehension, using `bag` or `set` (depending on whether or not the query filters out duplicate values). Omitting the optional `where` clause implies predicate E_p is the boolean value `true`;

   `select E from` $\mathtt{E}_1$ `as` $x_1, \ldots,$ $\mathtt{E}_n$ `as` x_n `where` $\mathtt{E}_p$
   $\mapsto$ $\{\!\{$`E |` $x_1 \leftarrow \underline{\mathtt{E}_1}, \ldots, x_n \leftarrow \underline{\mathtt{E}_n}$`,` $\mathtt{E}_p\}\!\}$

   `select distinct E from` $\mathtt{E}_1$ `as` $x_1, \ldots,$ $\mathtt{E}_n$ `as` x_n `where` $\mathtt{E}_p$
   $\mapsto$ `{E |` $x_1 \leftarrow \underline{\mathtt{E}_1}, \ldots, x_n \leftarrow \underline{\mathtt{E}_n}$`,` $\mathtt{E}_p$`}`

2. **join**

   The `join` construct is expressed as a conditional comprehension;

$\texttt{E}_1$ `as` x_1 `join` $\texttt{E}_2$ `as` x_2 `on` $\texttt{E}_c \mapsto \{\!\{\texttt{*} \mid x_1 \leftarrow \underline{\texttt{E}_1},\ x_2 \leftarrow \underline{\texttt{E}_2},\ \texttt{join}(x_1,\ x_2,\ \texttt{E}_c)\}\!\}$

with $\texttt{E}_c$ represents the join condition expression.

3. **group-by-having** The `group-by-having` construct resembles the SQL form, except that the one in QQL allows us to define an alias for each group-by criteria, noted as k_i.

$$\begin{aligned}
&\texttt{select E from } \texttt{E}_1 \texttt{ as } x_1, \ldots, \texttt{E}_n \texttt{ as } x_n \texttt{ where } \texttt{E}_p \\
&\texttt{group by } e_1 \texttt{ as } k_1, \ldots, e_r \texttt{ as } k_r \\
&\texttt{having } \texttt{E}_f \\
&\mapsto \{\!\{\texttt{E} \mid k_1, \ldots, k_r, \underline{(x_1, \ldots, x_n)} \leftarrow \\
&\qquad \{x_1 \leftarrow \underline{\texttt{E}_1}, \ldots, x_n \leftarrow \underline{\texttt{E}_n},\ \texttt{E}_p,\ \texttt{groupby}(k_1 \equiv e_1, \ldots, k_r \equiv e_r),\ \texttt{having}(\texttt{E}_f)\}\}\!\} \\
&= \{\!\{\texttt{E} \mid k_1, \ldots, k_r, \underline{(x_1, \ldots, x_n)} \leftarrow \\
&\qquad \{k_1, \ldots, k_r,\ \texttt{partition} \mid x_1 \leftarrow \underline{\texttt{E}_1}, \ldots, x_n \leftarrow \underline{\texttt{E}_n},\ \texttt{E}_p, \\
&\qquad\quad k_1 \equiv e_1, \ldots, k_r \equiv e_r,\ \texttt{partition} \equiv \\
&\qquad\quad \{o_1, \ldots, o_n \mid o_1 \leftarrow \underline{\texttt{E}_1}, \ldots, o_n \leftarrow \underline{\texttt{E}_n},\ e_1(\bar{o}) = e_1(\bar{x}), \ldots, e_r(\bar{o}) = e_r(\bar{x})\},\ \texttt{having}(\texttt{E}_f)\}\}\!\}
\end{aligned}$$

Group-by is based on the `partition` action [GS96]. The partition for each group contains all objects that have the same results for the grouping criterion. The `Having` clause defines a boolean predicate.

An internal construct, `group-by-default`, is introduced to create a default partition that carries all objects retrieved from query sources. It is used as the default grouping construct when aggregate functions are used in the `select` clause, but when there is no explicit group-by. The mapping is denoted as, and is consistent with, the case when the partition criteria $e_i(\bar{o}) = e_i(\bar{x})$ always returns `true`.

$$\begin{aligned}
&\texttt{select E from } \texttt{E}_1 \texttt{ as } x_1, \ldots, \texttt{E}_n \texttt{ as } x_n \texttt{ where } \texttt{E}_p \\
&\texttt{group by default} \\
&\mapsto \{\!\{\texttt{E} \mid \underline{(x_1, \ldots, x_n)} \leftarrow \{\ \{x_1, \ldots, x_n \mid x_1 \leftarrow \underline{\texttt{E}_1}, \ldots, x_n \leftarrow \underline{\texttt{E}_n},\ \texttt{E}_p\}\ \}\!\}
\end{aligned}$$

4. **aggregation** Aggregate functions perform operations on collections. For example `sum` implements operator `+`, while `min` and `max` implement value comparisons. In this part, we only discuss the group-by based aggregate functions; the field-access expression based aggregate methods are discussed in the **expression** part. These aggregate expressions can be directly mapped to primitive monoids defined in [GS96], where $\underline{\bar{x}}$ denotes a vector of query sources,

$$\begin{aligned}
&\texttt{sum E} \mapsto \texttt{sum } \{\texttt{E} \mid \bar{x} \leftarrow \underline{\bar{x}}\} \\
&\qquad \texttt{monoid sum = (0, id, +)} \\
&\texttt{max E} \mapsto \texttt{max } \{\texttt{E} \mid \bar{x} \leftarrow \underline{\bar{x}}\} \\
&\qquad \texttt{monoid max = (-}\infty\texttt{, id, } \lambda a \rightarrow \lambda b \rightarrow \texttt{a>b?a:b)} \\
&\texttt{min E} \mapsto \texttt{min } \{\texttt{E} \mid \bar{x} \leftarrow \underline{\bar{x}}\} \\
&\qquad \texttt{monoid min = (}\infty\texttt{, id, } \lambda a \rightarrow \lambda b \rightarrow \texttt{a<b?a:b)} \\
&\texttt{count E} \mapsto \texttt{count } \{\texttt{x} \mid \bar{x} \leftarrow \underline{\bar{x}}\} \\
&\qquad \texttt{monoid count = (0, (}\lambda x \rightarrow 1\texttt{), +)} \\
&\texttt{avg E} \mapsto \texttt{avg } \{\texttt{E} \mid \bar{x} \leftarrow \underline{\bar{x}}\} \\
&\qquad \texttt{avg can be derived from monoid sum and count: avg S = (sum S)/(count S)}
\end{aligned}$$

5. **order-by**

Let `E` be a `select-from-where` query, the `order-by` construct can be mapped as,

$$\begin{aligned}
&\texttt{E order by } e_1\ d_1, \ldots, e_n\ d_n \\
&\mapsto \{\!\{\texttt{x} \mid \texttt{x} \leftarrow \texttt{E},\ \texttt{orderby}(e_1\ d_1, \ldots, e_n\ d_n)\}\!\} \\
&\quad = \texttt{sorted}[e_1\ d_1, \ldots, e_n\ d_n]\ \{\!\{\texttt{x} \mid \texttt{x} \leftarrow \texttt{E}\}\!\}
\end{aligned}$$

where $e_i\ d_i$ defines the sorting criterion; `sorted[f]` can be implemented as in [GS96],

$$\texttt{sorted[f](x,y) = (f x)} \leq \texttt{(f y)?}\ merge[\mathcal{M}]\texttt{(x, y)} : merge[\mathcal{M}]\ \texttt{(y, x)}$$

6. **limit-offset** Let `E` be a `select-from-where` query, the `limit` and `offset` construct can be mapped as,

```
E limit n ↦ {{x | x ← E, limit n}} = take n {{x | x ← E}}
E offset n ↦ {{x | x ← E, offset n}} = drop n {{x | x ← E}}
```

where we implement `take` as a function based on pattern matching and basic monoid functions. Function `drop` can be implemented analogously.

```
let take n S =
  let rec take' n S R =
    if n = 0 then R else
      match S with
      |merge[M](unit[M]a, S') -> take' (n-1) S' merge[M](R, unit[M]a)
      |zero[M] -> raise exception
  in take' n S zero[M]
```

7. **expression** QQL expressions, such as field expressions and binary operation expressions, directly correspond to value projection and algebra expressions. Let θ represent binary arithmetic or comparison operators (+,-,×,/,<,=,>,≤,≠,≥).

```
E as x ↦ x ≡ E
E#field ↦ E#field
E.field ↦ E.field
E1#[x=>E2] ↦ {E2 | x ← E1}
E#aggregate () ↦ F[aggregate] {x | x ← E}

valof E ↦ E
itemsof E ↦ v ← E

E1 θ E2 ↦ E1 θ E2
not E ↦ ¬ E
E1 and E2 ↦ E1 ∨ E2
E1 or E2 ↦ E1 ∧ E2
E1 in E2 ↦ E1 in {x | x ← E2}
E is Null ↦ E is Null
E is not Null ↦ E is not Null
E1 between E2 and E3 ↦ E1 between E2 and E3
```

Note that aggregate methods in field expressions are denoted as `F[aggregate-name]`, which is only different in notation from the raw group-by based aggregate functions (e.g. `(sum E)`). The purpose of this notational difference is to distinguish aggregation based on group-by and field-expressions, since these two are translated into different SQL. The latter one (i.e. the field-access-expression based aggregate expression) is translated into a sub query.

Term `(valof E)` is mapped into `E` directly based on the fact that nullable values in SQL queries are treated the same as non-nullable values; `(itemsof E)` denotes extracting values from a collection.

To demonstrate the applicability of this mapping let us consider a QQL query, the one we have demonstrated in section 4.5.3 for avatar generation. The mapping is demonstrated as below, note that, `group-by` splits all `customer` into small groups according to the associated role,

```
select role#name as rname, (count cust) as cnt
from customer as cust
where cust#orders#count() > 10
group by cust#role as role]
↦ { rname ≡ role#name, cnt ≡ count {cust | cust ← cust}
    | role, cust ←
      { cust ← customer, (F[count]{v ← cust#orders}) > 10, groupby(role≡cust#role) }
    }
```

Even after mapping query constructs to equivalent monoid comprehensions, it is still difficult to express constructs such as navigational queries and nested structures in SQL. In the following, we describe the approach of resolving navigational query by appropriate joins and sub-queries, and

flattening nested structures by adding object identity/key attributes, and demonstrate this process based on the query above. The full step-by-step translation is shown in Figure 4.23.

Step 1: Resolution of aggregate expressions

Aggregate expressions are tightly coupled with the group-by action. The semantics of group-by in SQL imposes the restriction that the return values of group-by queries can only be those referred to in the group-by criteria, while QQL allows us to define queries beyond this restriction by using navigational queries, that is, returning additional values by following the references of objects referred to in the group-by criteria.

For example, the query to find the department manager and the sum of salaries spent for each department can be specified in QQL by grouping `employees` by the `department` and then performing an aggregate calculation for the sum of `salaries`, and obtaining the `manager` entity by following the reference in `department` entity:

```
select dept#manager as manager, (sum emp#salary) as salaries
from employee as emp
group by emp#department as dept
```

Such a query cannot be translated directly into SQL, since SQL does not allow returning values from columns that are not referred to in the group-by clause. However, the translation is achievable by shifting aggregate expressions to sub-queries, as in:

```
select dept#manager as manager, v as salaries
from (select dept, (sum emp#salary) as v
      from employee as emp
      group by emp#department as dept)
```

The above transformed query performs the group-by and aggregation function in a sub query, which returns the group-by criteria `dept` and the aggregation result `v`. Then, using these return values, it returns `dept`'s `manager` in a navigational querying style. Now this transformed query can be mapped to equivalent SQL, except for expressions involving object associations, such as `dept#manager`. We discuss how to resolve object associations as a join expression in the next step.

More formally, the transformation can be expressed as below (c.f. Figure 4.23, step1), with an additional shifting step. The first step is to moving aggregation expressions into a sub-query that is placed to the right side of the comprehension. The second step is to shift the expression used as a parameter of the aggregate function to the right side of the sub-comprehension (i.e. the one corresponding to the sub-query). This step is used to prepare for resolving object associations in the aggregate expression:

$$
\begin{aligned}
&\{\!\{\texttt{k\#x, aggregate } \{\texttt{c}_j\texttt{\#e} \mid \bar{c} \leftarrow \underline{\bar{c}}\} \mid \texttt{k},\underline{\bar{c}} \leftarrow \{\texttt{c}_1 \leftarrow \underline{c_1}, \ldots, \texttt{c}_n \leftarrow \underline{c_n}, \texttt{ groupby(k} \equiv \texttt{c}_i\texttt{\#p)}\}\}\!\} \\
&= \{\!\{\texttt{k\#x, t } \mid \texttt{(k,t)} \leftarrow \\
&\quad \texttt{\{k, aggregate } \{\texttt{c}_j\texttt{\#e} \mid \bar{c} \leftarrow \underline{\bar{c}}\} \mid \texttt{k},\underline{\bar{c}} \leftarrow \{\texttt{c}_1 \leftarrow \underline{c_1}, \ldots, \texttt{c}_n \leftarrow \underline{c_n}, \texttt{ groupby(k} \equiv \texttt{c}_i\texttt{\#p)}\}\}\!\} \\
&= \{\!\{\texttt{k\#x, t } \mid \texttt{(k,t)} \leftarrow \\
&\quad \texttt{\{k, aggregate } \{\texttt{v} \mid (\bar{c},\texttt{v}) \leftarrow \underline{(\bar{c}, v)}\} \mid \\
&\qquad \texttt{k,}\underline{(\bar{c}, v)} \leftarrow \{\texttt{c}_1 \leftarrow \underline{c_1}, \ldots, \texttt{c}_n \leftarrow \underline{c_n}, \texttt{ v} \equiv \texttt{c}_j\texttt{\#e, groupby(k:c}_i\texttt{\#p)}\}\}\!\}
\end{aligned}
$$

For concreteness, we show the resolution of the aggregation expression in the above query. The first step of resolution is to move the aggregation into a sub-query, which splits all employees into small groups according to the `department` and, for each group, returns the `department` and the sum of `salary`. The second step is to shift the expression (i.e. `emp#salary`) of the aggregate function parameter to the right side of the sub-query and, in this example, we bind this expression to a fresh varible `v` that is returned with `emp`.

$$
\begin{aligned}
&\{\!\{\texttt{dept\#manager, sum } \{\texttt{emp\#salary} \mid \texttt{emp} \leftarrow \underline{\texttt{emp}}\} \mid \\
&\quad \texttt{dept, } \underline{\texttt{emp}} \leftarrow \{\texttt{emp} \leftarrow \underline{\texttt{employee}}\texttt{, groupby(dept} \equiv \texttt{emp\#department)}\}\}\!\} \\
&= \{\!\{\texttt{dept\#manager, t } \mid \texttt{(dept,t)} \leftarrow \\
&\quad \texttt{\{dept, sum } \{\texttt{emp\#salary} \mid \texttt{emp} \leftarrow \underline{\texttt{emp}}\} \mid \\
&\qquad \texttt{dept,}\underline{\texttt{emp}} \leftarrow \{\texttt{emp} \leftarrow \underline{\texttt{employee}}\texttt{, groupby(dept} \equiv \texttt{emp\#department)}\}\}\!\} \\
&= \{\!\{\texttt{dept\#manager, t } \mid \texttt{(dept,t)} \leftarrow \\
&\quad \texttt{\{dept, sum } \{\texttt{v} \mid (\texttt{emp,v}) \leftarrow \underline{(\texttt{emp, v})}\} \mid \\
&\qquad \texttt{dept,}\underline{(\texttt{emp, v})} \leftarrow \{\texttt{emp} \leftarrow \underline{\texttt{employee}}\texttt{, v} \equiv \texttt{emp\#salary, groupby(dept} \equiv \texttt{emp\#department)}\}\}\!\}
\end{aligned}
$$

Step 2: Resolution of object associations

The path expression $v_1\#\ldots\#v_n$ indicates a join sequence when it includes a v_{i+1} that is a class reference from v_i path expression. The object association is resolved as a join expression. Suppose orm entity v_i contains a reference to orm entity v_{i+1}. Then field-access expression $v_i\#v_{i+1}$ can be resolved as below (c.f. Figure 4.23 step 2)

$\mathtt{v}_i\mathtt{\#v}_{i+1} \mapsto \mathtt{t} \leftarrow \underline{\mathtt{typeof}(\mathtt{v}_i\mathtt{\#v}_{i+1})}, \mathtt{join}(\mathtt{v}_i, \mathtt{t})$

where `typeof` returns the type of expression expression.

The `join` predicate is used with the intended meaning that `join(x, t)` should be present only if there is an object reference from object `x` to object `t`. This implies that the path expression chain only has to be resolved until the first non-class type is encountered. Note that this `join` predicate doesn't have an explicit user-defined condition expression. It is defined based on the foreign key information embedded in the orm class definition.

Let $\mathrm{R}_i = \mathrm{typeof}(\mathrm{v}_1\#\ldots\#\mathrm{v}_i)$ be the type of the successive object references in prefixes of its argument. Consider the expression `car#driver#address#street`, Then the full expression can be resolved as:

R_0 = typeof(car) = *car*
R_1 = typeof(car#driver) = *driver*
R_2 = typeof(car#driver#address) = *address*
R_3 = typeof(car#driver#address#street) = *string*

$\mathtt{car\#driver\#address\#street} \mapsto$
$\mathtt{c} \leftarrow \underline{\mathtt{car}}, \mathtt{d} \leftarrow \underline{\mathtt{driver}}, \mathtt{join(c,d)}, \mathtt{a} \leftarrow \underline{\mathtt{address}}, \mathtt{join(d,a)}, \mathtt{s} \equiv \mathtt{a\#street}$

Step 3: Resolution of nested structures

A query like:

```
select u, u#orders
from user as u
```

will not return a simple collection of `order` objects. Instead it will return a collection of collections of `order` objects. Each inner collection will be the `order` objects from a single user.

The return of such nested structures is resolved by adding additional object identity/key information and shifting the condition that enforces respecting the nesting structure to the outer level. This process starts from the outer-most level and is applied until all returned nested structures are flat. The key information is used to re-construct the nested structure from the returned records. For example, the `order`s collections can be re-constructed from a flattened result by `user`'s `id`,

$\mathtt{\{\{id = 1, od = order}_1\mathtt{\}, \{id = 2, od = order}_2\mathtt{\}, \{id = 2, od = order}_3\mathtt{\}\}} \Rightarrow$
$\mathtt{\{\{id = 1, ods = \{order}_1\mathtt{\}\}, \{id = 2, ods = \{order}_2\mathtt{, order}_3\mathtt{\}\}\}}$

The flattening of one nesting level can be described as follows, where function `key` returns the entity identifier, where $\mathtt{E}_p$ represents any further predicates obtained from the where clause.

$\mathtt{\{(x, \{y \mid y \leftarrow \underline{E}_y, join(x, y)\}) \mid x \leftarrow \underline{E}_x, E}_p\mathtt{\}}$
$\mathtt{= \{(x, key(x), y) \mid x \leftarrow \underline{E}_x, E}_p\mathtt{, y} \leftarrow \underline{\mathtt{E}_y}\mathtt{, join(x, y)\}}$

For our `user`, `orders` query above, this translates as follows, where $\mathtt{E}_p$, as it evaluates to `true` for this example, has been removed:

$\mathtt{\{(u, \{od \mid od \leftarrow \underline{order}, join(u, od)\}) \mid u \leftarrow \underline{user}\}}$
$\mathtt{= \{(u, key(u), od) \mid u \leftarrow \underline{user}, od \leftarrow \underline{order}, join(u, od)\}}$

Step 4: SQL translation

The last step of the translation is to generate SQL generated from the comprehension according to the object-relational mapping information.

Most of comprehension constructs are mapped to SQL straightforwardly. Note that

- the `join` predicate without explicit condition (i.e. `join(x, y)`) represents the join of two orm classes using the foreign key inforamtion embedded in the orm class definition. It is normally translated to a join clause over two relational tables using the foreign keys, except that in the case when `x` and `y` has a many-to-many association, this join predicate results in a join of three tables, the relational table for class `x` and `y` and the third table, which stores the relationship between `x` and `y`.
- the `join` predicate with an explicit condition expression (i.e. `join(x, y, p)`) is translated into a join clause using the specified condition expression.
- the field-expression based aggregation (e.g. `F[count] e`, for some generator `e`) is translated into a sub-query (c.f. Figure 4.23 step 4).

As an example, we illustrate this QQL translation process using the query mentioned above, see Figure 4.23. The first step resolves the aggregation expression by shifting it into a sub-monoid comprehension. The step resolves the chains of object associations: `Step2a` resolves the field access expression in the `where` clause. `Step2b` resolves the one in the `group-by` clause. Since there is no nested structures returned in this query, the third step is omitted. The last step shows the SQL derived from the monoid comprehension.

```
select distinct role#name as rname, (count cust) as cnt
from customer as cust
where cust#orders#count() > 10
group by cust#role as role]

↦ { rname ≡ role#name, cnt ≡ count {cust | cust ← {cust}}
     | role, {cust} ←
         { cust ← customer, (F[count] {v ← cust#orders}) > 10, groupby(role≡cust#role) }
     }

=step1 { rname ≡ role#name, cnt ≡ t | (role, t) ←
        { role, count {cust | cust ← cust} |
          role, cust ←
                {cust←customer, (F[count]{v←cust#orders}) > 10, groupby(role≡c#role)}}}

=step2a { rname ≡ role#name, cnt ≡ t | (role, t) ←
        { role, count {cust | cust ← cust} |
          role, cust ←
                {cust←customer, (F[count] {v←{od | od←order, join(cust,od)}}) > 10),
                                  groupby(role≡c#role)}}}

=step2b { rname ≡ role#name, cnt ≡ t | (role, t) ←
        { role, count {cust | cust ← cust} |
          role, cust ←
                {cust←customer, (F[count] {v←{od | od←order, join(cust,od)}}) > 10),
                                  r ← role, join(cust, r), groupby(role≡r)}}}

=step4 SELECT distinct rolename as rname, t as cnt
       FROM (SELECT r.roleid as roleid, r.rolename as rolename, count(cust.custid) as t
             FROM customers cust left join roles r on cust.roleid = r.roleid
             WHERE (SELECT count(od.orderid) FROM orders od WHERE od.custid = cust.custid) > 10
             GROUP BY r.roleid, r.rolename)
```

Figure 4.23: Translation of QQL example

4.10 Summary

In this chapter, we embed a compile-time type-safe, object-oriented query language in a functional language, OCaml. This query language embraces the functionality of SQL, and also introduces object-oriented features that tends to result in shorter and more intuitive queries by using implicit join, query composition and nested results.

In order to guarantee the compile-time type safety of the embedded query language, we propose the approach of using a type avatar, a dummy structure, to capture the type constraints of queries. In section 4.5, we present the details of type avatars and rules for their generation. Section 4.3 introduced and employed phantom types [LM99a, CH03, FP06] to model the type constraints of SQL aggregation functions, thus allowing these type-overloaded functions to be checked at compile time in OCaml, a language that does not support type overloaded functions naturally. Furthermore, this phantom encoding is extended to support the type checking of all query functions in section 4.5. In section 4.7 and 4.8, we propose a type inference algorithm based on the type equation and compile queries into a self-contained query structure, thus, variables can be incorporated in queries, or queries can be dynamically composed at runtime while still maintaining compile-time type safety.

Finally, we show in section 4.9 the translation of object-oriented queries into equivalent SQL by leveraging monoid comprehensions as the intermediate representation, thus making QQL effective in querying SQL based relational databases.

Chapter 5

Transactions

A database transaction consists of a logical unit of database operations that are guaranteed to be executed, if at all then as a whole, to process user requests for retrieving data or updating the database. This also involves isolating the operations of a transaction from those of other transactions proceeding in parallel, guaranteeing that the transaction will leave the database in a consistent state when it has completed and ensuring that, in spite of certain finite failures, the results of completed transactions can be recovered. In SQL, transactions have an explicit form, requiring transaction commands to be issued at the beginning and end of the transaction, i.e. issuing `BEGIN` before the query and update operations of the transaction, `COMMIT` after all operations have completed, or `ROLLBACK` if an operation has failed or an error has occurred and the operations that have been completed in the transaction so far must be undone.

The Java Database Connection (JDBC) [FEB03] and other libraries provide programming support for database transactions using the exception mechanism to manage database errors and invoke transaction rollbacks. Frameworks, like Spring [Joh02, JHA+05], rely on aspect-oriented programming (AOP) techniques [KLM+97, EAK+01] to separate and encapsulate the cross-cutting transaction handling into separate modules, which can then be applied to different parts of the program through a weaving process. These approaches involve a significant amount of repetitive code (in the case of a JDBC program) or complex, administrative configuration (in the case of the AOP approach) to implement transaction handling. Further, these programming approaches fail to provide a compile time protocol-safe solution that guarantees the necessary transaction is invoked in the right way and at the right place.

Taking advantage of functional programming language features, we investigate a higher-order transaction pattern that is simpler than previous approaches, does not involving additional run time processing or complex configurations for accomplishing transaction management, and ensures, at compile time that the transaction pattern is used in a protocol safe way.

5.1 The transaction problem

Database interactions need to be grouped together, enclosed and properly terminated, with either a commit or rollback, inside a transaction. In low level database interaction frameworks, such as Java with JDBC, it requires programmers to issue transaction commands and handle transaction exceptions in an explicit form, i.e. issuing `BEGIN` before the query and update operations of the transaction, `COMMIT` after all operations have completed, or `ROLLBACK` if an operation has failed and the error has been caught by the `try-catch-finally` pattern (c.f. Figure 5.1). Such an approach involves a significant amount of repetitive code, distracting programmers from the program logic. A common programmer error is to leave out necessary parts of the standard transaction template, resulting in unexpected, and unwelcome, transaction behaviour and/or memory leaks in runtime systems. It becomes complicated to manage database transactions, especially in large scale database applications, or to write correct single usecase functions which span multiple database sources. The latter case requires writing and invoking transactions in a nested style; any exception must trigger the rollback action on all involved databases, while a final successful commit must be

synchronised across all participating databases, usually via the *two-phase commit* algorithm [LS76].

```
Connection conn = DriverManager.getConnection(...);
try {
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    ...
    stmt.close();
    conn.commit();
} catch(SQLException e) {
    conn.rollback();
} finally {
    conn.close();
}
```

Figure 5.1: JDBC-style try-catch-finally transaction

To eliminate the repetitive "*boilerplate*" transaction management code patterns, ORM frameworks layered on top of JDBC, such as Hibernate, are commonly used in combination with the Spring Framework [Joh02, JHA+05] to provide an aspect-oriented programming (AOP) solution [KLM+97, EAK+01] to transaction management.

The use of AOP has proved useful in modularising cross-cutting concerns such as profiling, error-handling, and security in large-scale software systems [CKF+01, CKFS01]. AOP based applications are typically constructed by developing the base functional modules of the program and a set of *aspects* that encapsulate the cross-cutting concerns. An aspect includes two constructs: a *pointcut* that identifies a set of functions or code locations, indicating when and where to weave the functional modules into the application, and an *advice* that is the implementation of the concern and will be executed when a pointcut is reached. In order to compose functional modules and these aspects for the complete program behaviour, a *weaving* process is required, which can be performed on source code during compile time, or on byte code during load-time or run-time. The use of AOP results in the reduction or removal of repetitive, but often complex, administrative code from the main code base for the configuration of pointcuts locating and weaving processing [JHD+09].

These approaches, however, do not provide a concise solution to the problem of developing transaction management code, as both require writing either repetitive or complex "*boilerplate*" transaction management code or configuration specifications, and cannot be viewed as protocol safe. That is, errors, such as leaving out necessary parts or defining incorrect pointcut locations or aspect advice, in these patterns or invoking database operations outside a transaction scope cannot be identified at compile time but require explicit code inspection or testing at runtime.

5.2 Higher-order simple transactions

Functional programming languages have proven to be a natural host for AOP [TK03, MTY05, WCK06], because of their inherent support for higher-order functions and their power of data abstraction. In this and the following sections, we investigate the design of a compile-time protocol-safe interface to transaction handling that disallows programmers from getting the transaction pattern wrong. This interface consists of higher-order functions for *simple* transactions, on a single database source, and *multiple* transactions, spanning several database sources, and captures the fact that the transaction advice in the context of AOP is an *around* advice that starts transaction before, and commits/rolls-back the transaction after the method invocation.

5.2.1 Transaction interface

A *use case* [Coc01] is a description of a system's behaviour in response to a request that originates outside the system. Use case analysis has become a popular software engineering technique in decomposing large system designs into small manageable elements. A use case function implements

```
module Session : sig
  type session

  type t (* entity type *)
  type id (* entity id type *)

  val save{t} : session -> t -> unit
  val load{t} : session -> id -> t
  val update{t} : session -> t -> unit
  val delete{t} : session -> t -> unit

  type 'a query
  val query : session -> 'a query -> 'a

  module Factory : sig
    type sf
    val make : Config.t -> sf
    val default : unit -> sf
  end

  module Transaction : sig
    val txn : Factory.sf -> (session -> 'arg -> 'rtn) -> 'arg -> 'rtn
    val eval : Factory.sf -> (session -> 'rtn) -> 'rtn
  end
end
```

Figure 5.2: Snapshot of Session interface

a single use case, generally requires transactional access to a database and is usually executed within a single transaction. In web applications, a single web request typically triggers a single use case execution to handle that request.

In Qanat, a *session* encapsulates a database connection together with Qanat's data structures for handling object relational management and querying. Thus sessions manage the relationships of persistent entities and are responsible for conveying data between programs and underlying databases. As shown in the `Session` module (see Figure 5.2), operations like `save`, `load`, `update`, `delete` and query execution, can be invoked on an initialized session instance.

Every database operation must be executed within the context of a transaction. A single transaction typically executes multiple database operations, most often on the same database and, hence, using the same session object, but sometimes spannng multiple databases and therefore using multiple different session objects, one for each different database involved in the transaction.

A *session factory* is an object that encapsulates configuration information for sessions and can produce session objects as required. However, the `Factory` module does not provide a public interface to initialize sessions, as that would allow programmers to erroneously obtain sessions outside the context of any transaction. Instead, the initialization of sessions can only be invoked internally by other functions inside the `Session` module, such as transaction functions. In addition, factories play the role of a cache and a connection pool, which caches the created open sessions and returns them directly when another request occurs. As constructing connections is an operation requiring considerable resources, it is a performance critical issue to provide such caching and pooling facilities.

Qanat's *simple transaction* is designed as a function (`txn` or `eval`) that takes a session factory instance and a use case function, the latter taking a session instance and an optional value as arguments. The transaction function gets a session from the session factory and invokes the use case within the transaction of the obtained session. When the use case finishes, it closes the session (automatically persisting any changes back to the database) and commits the database transaction. If an exception is thrown out of the use case, then the transaction triggers a rollback and tidies up. It is worth noting that the closing of a session when a transaction finishes does not terminate

its encapsulated connection. Instead, the connection is returned to the session factory, from which the session is created. Such a cached open connection is reused in future sessions.

The caching of an open session and encapsulated database connection in the session factory helps reduce the cost of session construction. It also maintains the consistency of orm entities that are managed by the session instance but whose modifications have not yet been persisted into the database. These modifications will be written to the database when the transaction ends and closes the session. However, within a single transaction, multiple usecases can be invoked on the same session and the use of the session cache means that the orm entities in the session are available and consistent to all such usecases. Since the same open session is always returned from one factory, multiple nested transactions invoked on the same session factory (i.e. the same open session) are grouped into a large transaction scope. Thus, they can be committed or rolled back as a unit. The full discussion of Qanat transaction scope is given in Section 5.2.2.

Function *txn* or *eval* is used for use cases with one or zero additional parameters respectively. Transaction functions invoke the use case function with the session, obtained from the session factory, and the optional parameter. The session factory takes the responsibility of initialising session instances from the specified configuration. Two different cases occur when getting session instances from the session factory:

No open session in session factory: When the transaction function (either *txn* or *eval*) is invoked, it gets a session instance from the given session factory. If there is no open session in the factory, a new session instance is initialised and bound to the factory, then returned. Meanwhile, a database transaction is started on the session instance. Within this transaction, the use case function is invoked. When the operation finishes, the handler closes the session and commits the transaction, which implies automatically persisting any changes back to the database. If any exception happens during the execution and is not caught within the use case function, the transaction rolls back, then re-throws the exception for the upper-level exception handler.

Open session present in session factory: In the case when there is open session found in the session factory, i.e. the use case is being invoked within the context of running transaction, the open session will be returned and the use case function is invoked in the scope of the current database transaction. If an exception is thrown out of the use case function, the current handler re-throws it for the upper-level transaction handler to process.

Thus if a use case function is invoked, when there is currently no transaction in operation with respect to that session factory, a new transaction is begun, the use case is executed within this transaction and the transaction terminates when the use case is finished. On the other hand, if there was a transaction already in operation with respect to that session factory, the new use case is simply added to the current transaction.

Figure 5.3 gives the implementation of the simple transaction function, `txn`, in OCaml. The function `eval` is similar to `txn`, except that the use case used in *eval* doesn't have the additional parameter. The algorithm of this function is described as:

1. Obtain a session instance from the session factory, create a new session object if necessary.
2. Check the transaction status of the obtained session, start the transaction if it is in '*Idle*' status; throw exception if in the '*Failed*' status; carry on to the next step if in '*Open*' status.
3. Invoke the use case function with the given argument.
4. (a) Commit the transaction if it was started within the current function, and remove the closed session from the factory; otherwise, leave the transaction to be committed by upper level function (previous invocation of `txn`). Return the use case result.
   (b) Rollback the transaction if any exception occurs and the transaction is started within the current function; otherwise, let the upper level function rollback the transaction. Throw the exception to other exception handlers.

```
let txn : Factory.sf -> (session -> 'arg -> 'rtn) -> 'arg -> 'rtn =
  fun sf usecase arg ->
    let sess = Factory.curr_session sf in
    (* check if a new transaction needs to be started *)
    let handle_txn =
      match sess#txn_state with
      | 'TRANSACTION_IDLE -> (sess#begin_txn; true)
      | 'TRANSACTION_OPEN -> false
      | 'TRANSACTION_FAILED -> raise (Exception "Transaction failed")
    in
    try
      (* invoke usecase function *)
      let rtn = usecase sess arg in
      (* close transaction if it is started within this function *)
      if handle_txn then (
         sess#close; Factory.remove_curr_session sf)
      ) else ();
      rtn
    with exn -> (
         (* rollback transaction if it is started within this function *)
         if handle_txn then sess#rollback else ();
         raise exn
    )
```

Figure 5.3: Simple Transaction Function (txn) in OCaml

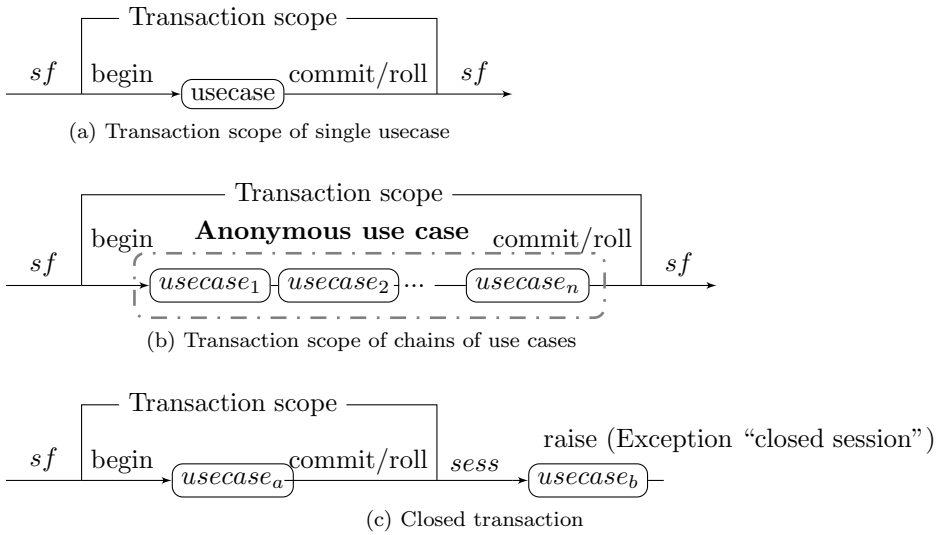
Higher-order functions provide a simple transaction pattern that invokes the use case function within the transaction automatically. The abstract of session type and no exposed initialisation function disallow session instances from being created except with an appropriate transaction context. Thus we have a compile-time protocol-safe way, which is also concise and simple for programmers, of ensuring that no use case can be executed on a new session instance without being executed inside a transaction.

That still leaves open the possibility of a use case being invoked on an *old* session instance after the transaction the created the session instance, and the use case that it was created for, has terminated. This can occur if the use case that executed inside the transaction then passes the session instance it obtains from the transaction out of the scope of the transaction. We discuss this situation in the next section.

5.2.2 Transaction scope

In this section, we discuss the scope of transactions. A transaction function invokes a (top-level) use case function. This top level use case function can call other use case functions, and these will all execute within the scope of the transaction. When the top level use case function returns, or is terminated by an exception, the transaction function will terminate the transaction and close the session. At this point the proper behaviour is that there should be no further references to the session instance that was used in the transaction, and the session instance can be garbage collected. However, it is possible for the use case function to pass a reference to the session instance out of its scope (e.g. by returning it as a return value or adding it to a global variable). With such a session instance, a use case function can be invoked without being in the context of an executing transaction. However, since the session instance in question is closed, any attempt to use the session instance to access or update the database will immediately cause a runtime error and an exception to be thrown.

1. In the case when there is no open session attached in the session factory (a typical situation), the invocation of use case using function `txn` or `eval` normally means to start a new transaction on the newly initialised session object, and commit/rollback the transaction when



(a) Transaction scope of single usecase

(b) Transaction scope of chains of use cases

(c) Closed transaction

Figure 5.4: Transaction scope

the use case finishes. The transaction scope is the same as that of the use case execution, `Transaction.txn sf usecase arg`, as depicted in Figure 5.4a.

2. It is also possible to group a set of use cases or to invoke one use case inside another one, such as the one demonstrated in Figure 5.4b, which constructs an anonymous use case by invoking a sequence of functions from $usecase_1$ to $usecase_n$. The invocation of this anonymous use case initialises the session object and starts the transaction. Because all of these use case functions work on the same session factory, it implies the same session object (either obtained from the session factory by using the transaction function, or passed from the previous use case in the chain) is used for the use case invocation, thus all of these use cases are located in the same transaction scope. Such an operation chain could be defined and invoked as:

```
Transaction.eval sf
  (* anonymous use case *)
  (fun sess ->
       usecase1 sess arg1;
       usecase2 sess arg2;
       ...
       usecasen sess argn)
```

3. Figure 5.4c depicts an illegal situation, in which $usecase_b$ is invoked directly by passing a session object that is returned from the execution of $usecase_a$. Because the session object is automatically closed after the use case execution in order to commit the modifications, the execution of $usecase_b$ (assuming it performs database operations) will raise a session-closed runtime exception.

5.2.3 Transaction isolation

In addition to the transaction functions and abstract session interface, the proposed transaction approach supports various transaction isolation levels [Amb03a]. That is, to avoid conflicts during different transactions, a DBMS uses lock mechanisms for blocking access by others to the data that is being accessed by the transaction. Once a lock is set, it remains in force until the transaction is committed or rolled back. For instance, a DBMS could lock a row of a table until updates to it have been committed. Such a lock prevents a user from executing a dirty read, that is, reading a value before it is made permanent. The transaction level determines how these locks are set, ranging from not supporting transactions at all to supporting transactions that enforce very strict access rules. The less strict levels do compromise the ACID properties of transactions, for instance,

allowing to read dirty data being modified by other transactions. However, for some situations, the compromises made are acceptable for the semantics of the application and provide a significant performance improvement.

Because the session instance can only be initialised from the session factory inside the transaction functions, the isolation functions are lifted to the session factory module. They are defined in OCaml using variant types, as follows:

```
type isolation = TRANSACTION_NONE | TRANSACTION_READ_COMMITTED
               | TRANSACTION_READ_UNCOMMITTED
               | TRANSACTION_REPEATABLE_READ | TRANSACTION_SERIALIZABLE

val set_isolation : Factory.sf -> isolation -> unit
val get_isolation : Factory.sf -> isolation
```

These isolation levels have the same semantics as the corresponding levels in databases, and will be set, in the database, through a low-level JDBC-like interface provided by the session instance after it is initialised from the session factory.

The semantics of these isolation levels, ranging from less strict to the most strict one, are defined as follows:

Read uncommitted allows dirty reads, specifying that statements can read rows that have been modified by other transactions but not yet committed.

Read committed states that query statements cannot read data that has been modified but not committed by other transactions. There are no dirty reads (reads of uncommitted data) when this isolation level is set.

Repeatable read specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction completes.

Serializable is the most strict isolation level. It specifies that all transactions occur in a completely isolated fashion; i.e. statements cannot read data that has been modified but not yet committed by other transactions; no other transactions can modify data that has been read by the current transaction until the current transaction completes; other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.

5.3 Higher-order multiple transactions

It is not uncommon for programs to interact with several data sources simultaneously. Such operations involve transactions on multiple databases, for instance, performing data migration between different systems. Since the simple transaction is incapable of managing such distributed transactions, we propose a slightly more complex higher-order multiple-transaction interface for this purpose, which includes a parameterised abstract data type that helps capture the status of transaction invocation.

Our aim here is to design a compile-time protocol-safe way of invoking use case functions that interact with multiple database sources (i.e. multiple sessions) within a distributed transaction. In particular, we want to use the types to ensure that all necessary sessions are covered by the transaction before the use case can be invoked.

As well as the type specific issues to support our compile-time protocol safety goals, there are a number of practical issues that need to be addressed for such a multiple transaction scheme, not least of which is that an atomic commitment protocol for the multiple transactions is necessary. For clarity of presentation, we first present and explain our approach in terms of a system that ignores the atomic commitment requirement, to better explain the type system in Sec. 5.3.1, and then show how the resulting system and types can be adapted to include atomic commitment in the form of the standard Two Phase Commit algorithm in Sec. 5.3.2.

5.3.1 Higher-order multiple transactions without atomic commit

As with simple transactions, use case functions for multiple transactions have two parameters: the session parameter and an additional use case parameter. In this case, the session parameter is a structure that holds all the session objects. We could use a list of sessions for this parameter, but then the type system would not be able to reject an invocation of the use case with the wrong number of sessions. We could instead use a tuple of sessions, but then we would need entirely different copies of the multiple transaction system, one specialised for 2-session use cases, one for 3-session use cases etc. Instead, by judicious use of higher order functions and currying, we can handle all multiplicities of sessions in one system if the session parameter is a right deep tree of session objects. This is not a list because the rightmost leaf is a session object, rather than an empty right deep tree. Thus the type of a multiple transaction use case function, `f`, is as follows:

$$\texttt{f : (sess}_1\texttt{ * (sess}_2\texttt{ * (sess}_3\texttt{ * ... * sess}_n\texttt{)...)) -> 'arg -> 'rtn}$$

where the first parameter is a right deep tree of session instances, and the second parameter is an argument to the use case function.

The multiple transaction type is the parameterised abstract data type t:

```
type ('a, 'b, 'c, 'd) t = (('a -> 'b) * (('c -> 'd) -> 'c -> 'd))
```

Transaction type t takes four type parameters, representing a pair of functions, the first one is a function from $'a$ to $'b$, the second one is a higher-order function that maps a function of type $'c \rightarrow 'd$ to another function of the same type.

To unravel this type, first note that $'c$ corresponds to the $'$`arg` parameter type of the use case function, and $'d$ corresponds to the $'$`rtn` return type of the use case function. Thus $'c \rightarrow 'd$ represents the curried use case function after it has been applied to the full session structure but before it has been applied to the additional use case argument.

The $('c \rightarrow' d) \rightarrow 'c \rightarrow 'd$ type corresponds to a higher order function that takes the curried use case function and returns a new function of the same type but surrounded by the appropriate transaction management code: begin the transaction, execute the parameter function, (i.e. the curried use case function) in a try block and then commit the transaction. If an exception is thrown, a rollback is issued instead of a commit.

That leaves the $'a \rightarrow 'b$ part of the type. Here $'b$ corresponds to the curried use case function as before (applied to the full session structure but not to the additional argument), whereas $'a$ corresponds to a suffix of the right deep tree session structure, i.e. the session structure remaining when some of the initial sessions have been (stack-like) popped off. Thus the function of type $'a \rightarrow 'b$ is the partially curried use case function that needs to be applied to the remainder of the session structure (of type $'a$) to return the use case function curried with the full session structure. Thus type t can be expressed in an alternative form as:

```
type ('remaining_sessions, 'curried_usecase, 'arg, 'rtn) t =
      (('remaining_sessions -> 'curried_usecase) * (('arg -> 'rtn) -> 'arg -> 'rtn))
```

The multiple transaction interface includes functions `txnK`, `txnN`, `txnT` and `txnExc` (Figure 5.5). Function `txnK`, `txnN` and `txnT` obtain or initialise a session instance from the session factory and pass this session instance into the (partially curried) use case function. This returns a function that takes the remaining required session instance pair as argument and returns a use case function curried with the full session structure, obtained by passing in the remaining session structure. In addition to this returned function, function `txnK`, `txnN` and `txnT` also construct a higher-order function inside which a transaction is started before, and committed/rolled back after, the execution of the argument function (i.e. the curried use case function).

- Function `txnK` is the starting point of the transaction construction, it takes a session factory and the plain use case function (not wrapped as type `t`) as parameters, and returns a value of type `t`. Since this is placed in most deeply nested centre of the use case execution expression, this function is called the '*kernel*' function.

```
(* parameterized transaction type *)
type ('a, 'b, 'c, 'd) t = (('a -> 'b) * (('c -> 'd) -> 'c -> 'd))

(* transaction apply, in start (kernel) status *)
txnK : Factory.sf -> ((session * 'b) -> 'c -> 'd) ->
                  ('b, ('c -> 'd), 'c, 'd) t

(* transaction apply, in non-terminal status *)
txnN : Factory.sf -> ((session * 'b),('c -> 'd), 'c, 'd) t ->
                  ('b, ('c -> 'd), 'c, 'd) t

(* transaction apply, in terminal status *)
txnT : Factory.sf -> (session,('c->'d),'c,'d) t -> ('c,'d,'c,'d) t

(* execute use case within multiple transactions *)
txnExc : ('a, 'b, 'a, 'b) t -> 'a -> `b
```

Figure 5.5: Multiple-transaction Interface

- Function `txnN` works in a similar way to `txnK`, except that it accepts a value of type `t` as the parameter. It returns two functions, one constructed by passing an obtained session instance to the first function of the passed value that is of type `t` and returned either from the execution of `txnK` or `txnN`, another one constructed by adding transaction handling around the execution of passed use case, which has already been wrapped by the transaction handling of previously passed session instances. This function is called a '*non-terminal*' function, since it adds session instances from the second one to the second last one.
- Function `txnT`, called a '*terminal*' function, returns the curried use case function and the higher-order function for invoking the last transaction handling.
- Function `txnExc` is used to invoke the use case execution with a specified argument.

Unlike the case with simple transactions, attempting to start a multiple transaction while there is any other transaction, simple or multiple, in progress is not allowed and will trigger a runtime exception. The problem is that the simple semantics in the case of a simple transaction joining an ongoing simple transaction on the same database, does not extend to the more complex situations involved with multiple transactions. While a consistent semantics may be possible for such cases, it appears that such a semantics is sufficiently complex to make it a unsuitable for a programmer friendly framework.

Figure 5.6 gives an implementation of these multiple-transaction functions in OCaml.

Assume use case function `f` works on $session_a$, $session_b$, $session_c$, defined in module $\texttt{Txn}_A$, $\texttt{Txn}_B$ and $\texttt{Txn}_C$ respectively, and one additional parameter α. We denote the corresponding session factories as sf_a, sf_b, and sf_c. The following code invokes f within multiple transaction scopes:

```
val f : (session_a * (session_b * session_c)) -> α -> β

Txn_C.txnExc (Txn_C.txnT sf_c (Txn_B.txnN sf_b (Txn_A.txnK sf_a f))) α
```

For concreteness, we show the type of the returned value by applying the transaction function step by step. Symbol $\equiv$ denotes the real type represented by $('a,'b,'c,'d)\ t$.

```
f : (session_a * (session_b * session_c)) -> α -> β

(Txn_A.txnK sf_a f) : ((session_b * session_c), (α -> β), α, β) t
            ≡ (((session_b * session_c) -> (α -> β)) * ((α -> β) -> α -> β))

(Txn_B.txnN sf_b (Txn_A.txnK sf_a f)) : (session_c, (α -> β), α, β) t
            ≡ ((session_c -> (α -> β)) * ((α -> β) -> α -> β))

(Txn_C.txnT sf_c (Txn_B.txnN sf_b (Txn_A.txnK sf_a f))) : (α, β, α, β) t
            ≡ ((α -> β) * ((α -> β) -> α -> β))

(Txn_C.txnExc (Txn_C.txnT sf_c (Txn_B.txnN sf_b (Txn_A.txnK sf_a f))) α) : β
```

Note that each time we apply function `txnK`, `txnN` or `txnT`, the transaction function initialises a session instance from the session factory but without starting a new transaction on this session instance,

When all required session instances are fed into the use case function (i.e. after the invocation of function $\texttt{Txn}_C\texttt{.txnT}$ in this example), function `txnExc` applies the obtained higher-order function (i.e. the second function in type `t`), which includes transaction handling for all obtained session instances, on the curried use case function (i.e. the first function). This instructs all required transactions to start before the use case execution, and commit or roll back (if an exception is thrown from the use case) after the use case execution.

Figure 5.7 gives an example showing how to synchronize the price of a produce in three different databases. The final price is determined by the minimum price of the product. The usecase function `synchronize_price` loads product instances from session `sess01`, `sess02` and `sess03`, calculates the minimum price, and then sets the price for these loaded product instances. We first initialize session factories from various config files, then invoke the usecase through the multiple transaction interface by passing these session factories (`sf01`, `sf02` and `sf03`) to `synchronize_price` one by one. This higher-order transaction interface ensures that necessary transactions are invoked before and committed after the execution of the usecase function.

We demonstrate here that the multiple-transaction interface is consistent with the simple transaction function, `txn`. Suppose `apply` is a higher-order, polymorphic function that performs function `f` on the given parameter `arg`, results in data of type `b`; `wrap` converts a function with type $'a \rightarrow' b \rightarrow' c$ to type t.

```
let apply : ('a -> 'b) -> 'a -> 'b = fun f -> fun arg -> f arg

let wrap : ('a -> 'b -> 'c) -> ('a, ('b -> 'c), 'b, 'c) t = fun f -> (f, apply)
```

where $('a, ('b \rightarrow' c), 'b, 'c)\ t = ('a \rightarrow ('b \rightarrow' c)) * (('b \rightarrow' c) \rightarrow' b \rightarrow' c)$ and `f` is of type $'a \rightarrow' b \rightarrow' c$.

A given function, $f_{usecase}$: $session \rightarrow' arg \rightarrow' rtn$, can be expressed as data of type t by using $wrap\ f_{usecase}$. Following the definition of `txnT` and `txnExc`, $f_{usecase}$ could be invoked as

```
Transaction.txnExc (Transaction.txnT sf (wrap f_usecase)) arg
```

In general, a simple transaction function *txn* could be defined as

```
let txn = fun sf -> fun f -> fun arg -> txnExc (txnT sf (wrap f)) arg
```

5.3.2 Higher-order multiple transactions with atomic commit

We use an implementation of the standard two phase commit algorithm to ensure that the full nest of transactions on different databases can atomically commit.

In the two phase commit algorithm [LS76], transaction commits on multiple databases are separated into two phases, the *voting* phase and the *completion* phase. During the first phase, all transaction handlers prepare for the transaction commit (by flushing all dirty data to disk and writing appropriate messages to the database log, and return a vote indicating that whether the global transaction can be committed from their individual point of view, or must be rolled back. Based on the voting obtained from all involved transactions, the coordinator (the higher-order

```
let txnK : Factory.sf -> ((session * 'b) -> 'c -> 'd) -> ('b, 'c -> 'd, 'c, 'd) t =
  fun sf f ->
    let sess = Factory.curr_session sf in
      ((fun b -> f (sess, b)),
       (fun ff args ->
          (match sess#txn_state with
          | `TRANSACTION_IDLE -> sess#begin_txn
          | _ -> raise (Exception "Transaction failed"));
          try
            let rtn = ff args in
            sess#flush;
            sess#commit;
            sess#close;
            Factory.remove_curr_session sf;
            rtn
          with exn -> sess#rollback; raise exn))

let txnN : Factory.sf -> ((session * 'b), 'c -> 'd, 'c, 'd) t -> ('b, 'c -> 'd, 'c, 'd) t =
  fun sf (f, fcurry) ->
    let sess = Factory.curr_session sf in
      ((fun b -> f (sess, b)),
       (fun ff args ->
          (match sess#txn_state with
          | `TRANSACTION_IDLE -> sess#begin_txn
          | _ -> raise (Exception "Transaction failed"));
          try
            let rtn = fcurry ff args in
            sess#flush;
            sess#commit;
            sess#close;
            Factory.remove_curr_session sf;
            rtn
          with exn -> sess#rollback; raise exn))

let txnT : Factory.sf -> (session, 'c -> 'd, 'c, 'd) t -> ('c, 'd, 'c, 'd) t =
  fun sf (f, fcurry) ->
    let sess = Factory.curr_session sf in
      ((f sess),
       (fun ff args ->
          (match sess#txn_state with
          | `TRANSACTION_IDLE -> sess#begin_txn
          | _ -> raise (Exception "Transaction failed"));
          try
            let rtn = fcurry ff args in
            sess#flush;
            sess#commit;
            sess#close;
            Factory.remove_curr_session sf;
            rtn
          with exn -> sess#rollback; raise exn))

let txnExc : ('a, 'b, 'a, 'b) t -> 'a -> 'b =
  fun (f, fcurry) args -> fcurry f args
```

Figure 5.6: Multiple Transaction Functions in OCaml

```
let synchronize_price (sess01, sess02, sess03) id =
  let p01 = Session.load{product} sess01 id in
  let p02 = Session.load{product} sess02 id in
  let p03 = Session.load{product} sess03 id in
  let price = min (min p01#price p02#price) p03#price in
  p01#set_price price;
  p02#set_price price;
  p03#set_price price

let _ =
  let sf01 = Session.Factory.make config01 in
  let sf02 = Session.Factory.make config02 in
  let sf03 = Session.Factory.make config03 in
  Transaction.txnExc (Transaction.txnT sf03
       (Transaction.txnN sf02 (Transaction.txnK sf01 synchronize_price))) 100
```

Figure 5.7: Example of using multiple transaction

transaction function in our case) instructs all involved transactions to complete the same action, either commit, if all votes were unanimously for commitment, or to roll back otherwise.

Our design for multiple transactions involves invoking the sub transactions in a nested style, and the transaction action is added to the use case function through constructing a higher-order function, inside which the transaction handling code is placed around the execution of use case function. Then these higher-order functions are called in a nested style to construct a function that includes transaction handling for all involved databases and executes the use case. Such an approach is achieved by using a parameterised type $('a,'b,'c,'t)t$ as discussed in the previous section.

In this section, we extended type `t` to to accumulate transactions' two-phase-commit voting and commit and rollback actions, which are represented as two separate functions for each involved transaction in the new type `'x t2pc` (the type for two-phase commit):

```
type 'x t2pc = ('x * (bool * (unit -> unit) * (unit -> string)))

type ('a, 'b, 'c, 'd) t = (('a -> 'b) * (('c -> 'd) -> 'c -> 'd t2pc))
```

`'x` represents the type of the use case return value, The calculated vote for `commit` or `rollback` from inner transactions is represented with type `bool`, $unit \rightarrow unit$ represents the operation of committing the prepared transaction, $unit \rightarrow string$ represents the operation of roll back, where the string return type is for returning an error message about what caused the roll back. These commit and rollback function types act as accumulators to build the nested commit and rollback function that will commit or rollback the whole stack of individual database transactions.

In this new type `t`, the higher-order function (the second element of the pair in `t`) adds transaction code around the execution of the use case (as before) and prepares the commit, then returns the vote (to commit or to rollback), as well as a `commit` and a `rollback` function, which, if invoked, will commit or rollback all the lower level transactions in a nested call. Then, at the top level, we get the vote decision of these transactions, and a commit and a rollback function to invoke commit/rollback on the full stack of nested database transactions.

Because of the data abstraction employed, these extensions over the non-atomic commit case are invisible to users, thus we get a series of two-phase-commit transaction functions that have the same interface with the one we proposed in the previous section. Figures 5.8 and 5.9 present the implementation of the two-phase-commit, multiple transaction functions. These functions `txnK`, `txnN`, `txnT` and `txnExc` have the same meaning as the corresponding ones we proposed before, except that these functions don't commit the transaction directly when the use case execution finishes, instead, they prepare the transaction commit and return a vote to *commit*, if the database is ready to commit, or to *rollback* if any error occurs. In the case of *rollback* voting, the error information is returned as well.

Because multiple database transactions are started in a nested style, the voting of these transactions are passed and accumulated from the inner level to the outer level. Here we use '*accumulation*' to mean the process that the combination of two votes, i.e. *commit* and *commit* returns *commit*, *rollback* is returned in other cases. In addition to the accumulation of commit voting, the commit and rollback actions are accumulated together. Thus at the top level, a voting decision and a commit and a rollback function for all nested transactions are returned. Then based on this voting decision, `txnExc` invokes the commit or rollback operations on all involved database transactions.

It is worth noting that there are two different kinds of situations that can trigger the rollback of the global transaction. These two kinds of situations roll back the multi-database transaction in two different ways:

1. After the use case finishes, the different databases are asked to prepare to commit. One of the databases may not be able to commit and therefore votes to *rollback*, which triggers the rollback of all database transactions using the *rollback prepared* command.

2. Another case is when an exception is thrown from within the use case, before the use case had come to its natural conclusion. At this point, none of the databases have been asked to prepare to commit.

   Since the use case has been invoked from with a nested stack of transaction functions, one for each database involved in the multi-database transaction, the exception is caught by the deepest nested transaction on the stack, which triggers a normal rollback on its database, and re-throws the exception to the transaction above it. The exception filters up to the top level causing all the component transactions to roll back on the way.

Similarly, we demonstrate the simple transaction function `txn` can be expressed using these two-phase-commit transaction functions. Function `wrap` is the same as the one used in the previous section; while function `apply` is modified to simulate the higher-order function that returns additional two-phase-commit voting and functions. In the following code, we define the `commit` and `rollback` function as dummy functions, it is safe, since there is no transaction started when wrapping the use case function into the form of type `t`. The only transaction is started in `txnT`, and the transaction commit or roll back is invoked in `txnExc`. Using `wrap` and `apply`, simple transaction function `txn` can be defined as,

```
let apply : ('a -> 'b) -> 'a -> 'b t2pc =
    fun f -> fun arg -> (f arg), (true, (fun () -> ()), (fun () -> "dummy string"))

let wrap : ('a -> 'b -> 'c) -> ('a, ('b -> 'c), 'b, 'c) t = fun f -> (f, apply)

let txn = fun sf -> fun f -> fun arg -> txnExc (txnT sf (wrap f)) arg
```

5.4 Exclusive Transactions

Certain programming mistakes with respect to transactions can lead to problems that are hard to test for and hard to debug. If, for such cases, we can not make the type checker identify these problems at compile time, and if we cannot design the interface to make coding such errors impossible, then we take the approach that we should at least detect such errors at run time and fail as early and clearly as possible.

One class of mistakes that fall into this category, is that of executing more than one, uncoordinated, transaction at the same time from within the same thread of execution. There is, of course, no problem with running multiple uncoordinated transactions simultaneously from within different threads of execution. An example scenario that demonstrates the issue is when the programmer executes a use case function u_1 on session s_1 under transaction t_1. Within u_1, a new use case function u_2 is called on session s_2 under transaction t_2. Note that if $s_1 = s_2$, there is no problem, as the transaction function would identify that there is currently a transaction under way on s_1 and u_2 would merely join transaction t_1 and no new transaction would be created. However, if s_1 and s_2 were different, i.e. sessions on different databases, then two uncoordinated transactions would be in effect simultaneously in the same thread of execution. If t_2 commits successfully and

```
let txnK : Factory.sf -> ((session * 'b) -> 'c -> 'd) -> ('b, 'c -> 'd, 'c, 'd) t =
  fun sf f ->
    let sess = Factory.curr_session sf in
    ((fun b -> f (sess, b)),
     (fun ff args ->
        (match sess#txn_state with
        | `TRANSACTION_IDLE -> sess#begin_txn
        | _ -> raise (Exception "Transaction failed"));
        let rtn = try
                     let v = ff args in sess#flush; v
                  with exn -> sess#rollback; raise exn
        in
        let (vote, error_info) =
            try
                (sess#prepare_commit; (true, None))
            with exn -> (false, (Some (Printexc.to_string exn)))
        in
        (rtn, (vote, (fun () -> sess#commit_prepared;
                                sess#close;
                                Factory.remove_curr_session sf),
                     (fun () -> sess#rollback_prepared;
                                sess#close;
                                Factory.remove_curr_session sf;
                                match error_info with
                                |Some err -> err
                                | None -> "No error info") )))
    )

let txnN : Factory.sf -> ((session * 'b), 'c -> 'd, 'c, 'd) t -> ('b, 'c -> 'd, 'c, 'd) t =
  fun sf (f, fcurry) ->
    let sess = Factory.curr_session sf in
    ((fun b -> f (sess, b)),
     (fun ff args ->
        (match sess#txn_state with
        | `TRANSACTION_IDLE -> sess#begin_txn
        | _ -> raise (Exception "Transaction failed"));
        let (rtn, (inner_txn_vote, inner_commitfun, inner_rollbackfun)) =
           try
              let v = fcurry ff args in sess#flush; v
           with exn -> sess#rollback; raise exn
        in
        let (curr_txn_vote, error_info) =
           try
              sess#prepare_commit; (true, None)
           with exn -> (false, (Some (Printexc.to_string exn)))
        in
        (rtn, ((inner_txn_vote & curr_txn_vote),
              (fun () -> inner_commitfun ();
                         sess#commit_prepared;
                         sess#close;
                         Factory.remove_curr_session sf),
              (fun () -> let inner_err_info = inner_rollbackfun () in
                         sess#rollback_prepared;
                         sess#close;
                         Factory.remove_curr_session sf;
                         match error_info with
                         | Some err -> inner_err ^ err
                         | None -> inner_err ^ "No error info") )))
    )
```

Figure 5.8: Two-phase-commit Multiple Transaction Functions in OCaml

```
let txnT : Factory.sf -> (session, 'c -> 'd, 'c, 'd) t -> ('c, 'd, 'c, 'd) t =
  fun sf (f, fcurry) ->
    let sess = Factory.curr_session sf in
    ((f sess),
     (fun ff args ->
        (match sess#txn_state with
        | `TRANSACTION_IDLE -> sess#begin_txn
        | _ -> raise (Exception "Transaction failed"));
        let (rtn, (inner_txn_vote, inner_commitfun, inner_rollbackfun)) =
          try
            let r = fcurry ff args in (sess#flush; r)
          with exn -> sess#rollback; raise exn
        in
        let (curr_txn_vote, error_info) =
          try
            sess#prepare_commit; (true, None)
          with exn -> (false, (Some (Printexc.to_string exn)))
        in
        (rtn, ((inner_txn_vote & curr_txn_vote),
               (fun () -> inner_commitfun ();
                          sess#commit_prepared;
                          sess#close;
                          Factory.remove_curr_session sf),
               (fun () -> let inner_err_info = inner_rollbackfun () in
                          sess#rollback_prepared;
                          sess#close;
                          Factory.remove_curr_session sf;
                          match error_info with
                          | Some err -> inner_err ^ err
                          | None -> inner_err ^ "No error info") )))
    )

let txnExc : ('c, 'd, 'c, 'd) t -> 'c -> 'd =
  fun (f, fcurry) args ->
    let (rtn, (vote, commitfun, rollbackfun)) = fcurry f args in
    match vote with
    | true -> commitfun (); rtn
    | false -> let err_info = rollbackfun () in
               raise (Exception err_info)
```

Figure 5.9: Two-phase-commit Multiple Transaction Functions in OCaml (cont.)

terminates, it is still possible for t_1 to fail and try to roll back. However, then we have the problem that we can no longer roll back transaction t_2. Clearly, the programmer should have used a multiple transaction instead of two simple transactions. A similar, but more complicated scenario is possible with two or more nested multiple transactions or mixed simple and multiple transactions.

We take a simple approach to handling this problem. We use a global flag in the thread of execution, which is set when a transaction, simple or multiple, is initiated and cleared when it terminated. The transaction functions can thus determine if there is another transaction in effect and throw an exception if it is not possible for this new transaction to join the old. There is a slight complication with multiple transactions, as these consist of a number of separate coordinated transactions. However, adding a global transaction identifier and using a simple protocol is sufficient to manage all possible cases correctly.

5.5 Summary

We began this chapter by introducing the transaction approach used in JDBC and aspect-oriented programming. This allowed us to identify problems with some current approaches to transaction handling. JDBC's try-catch-finally mechanism provides a clean approach to committing or rolling back transactions when the transaction finishes or exception occurs. However it is also common for programmers to make mistakes using this this pattern that lead to incorrect behaviour in failure cases. Even if the pattern is used correctly, the result is considerable obscuring of the business logic of the program through polluting the code base with large amounts of transaction handling code.

Aspect-oriented transaction approach helps reduce both of these problems, but its external XML based configuration still leaves common error cases open, such as a reliance on runtime, instead of compile-time type checking, complex configuration issues and easily overlooked errors due to a reliance on coding and function naming conventions.

We use higher-order functions and data abstraction to provide a protocol-safe and concise transaction interface. In this chapter, we proposed interfaces for simple transaction that works on single database source, and multiple transactions that span more than one database sources. The higher-order feature allows programmers to write use cases in separate function, then easily use our proposed transaction interface to invoke the use case within a transaction scope, which automatically commits when the use case execution finishes, and rolls back when any exception occurs. With our proposed approach, we achieve a modular separation between use case functions and the transaction functions.

Chapter 6

The Qanat framework

A large number of approaches have been developed to simplify construction of, and to reduce errors in, data-driven applications. In the previous chapters, we took a different approach from those previously proposed, based on strict type checking at compile time, type inference, object relational mapping, and loosely coupled database interaction. We show that this approach is practical and effective by implementing a compile-time type-safe object relational framework, called Qanat, in the OCaml programming language with a loosely coupled SQL database.

This chapter is organised as follows: in section 6.1, we introduce the basic functionality and structure of Qanat framework. Then in section 6.2, we show how to program in Qanat by introducing an interactive bookshop platform that demonstrates most features of Qanat and give an informal description of the QQL query language. In section 6.3, we give the formal grammar of the Qanat language extensions, including the ones for directly embedding ORM mapping metadata and Qanat query language in the program.

6.1 Qanat introduction

Qanat is a compile-time type-safe, Hibernate-like ORM framework, implemented in the OCaml programming language.

Like Hibernate, Qanat provides ORM features, including both navigational and set oriented transactional database interaction, with a loosely coupled relational database management system (DBMS). By loosely coupled, we mean an independent DBMS, in our case an SQL DBMS, that may be simultaneously providing database services to other applications that do not use the Qanat framework. By contrast, a tightly coupled solution allows the language full and exclusive control over the database, considerably simplifying the type management issues but restricting the applicability of the result to niche, rather than main-stream cases.

Where Qanat differs from Hibernate, aside from in the host language, is in achieving compile-time type safety. Hibernate uses dynamic loading, run time reflection and byte code compilation in its implementation, all of which make compile-time type safety problematic. Our approach leverages OCaml's type inference facilities through the use of a mechanism we call *Type Avatars*, phantom types, higher order functions and exceptions to cleanly handle transactions, and program transformation using the CamlP4 preprocessor [dR03] to add type safe language extension support for querying and object mapping.

Qanat consists of a Log4J-like [Gul03] logging library, a low level but complete PostgreSQL interface library, the main Qanat library and a set of tools. The PostgreSQL library was required as currently available PostgreSQL libraries for OCaml provide only limited support for transactions and for query parameter escaping and they force loading entire query results eagerly from the database, even if these results are very large, rather than meeting our requirements of loading results in blocks, lazily on demand. The main library defines the language extensions used during compilation of OCaml programs using Qanat as well as required modules and utility functions. The tool set includes:

DBSchema: A program to extract the schema from a PostgreSQL database and output a Qanat

specific schema description file in JSon format [Cro06].

ORMGen: A program to generate the Qanat-OCaml class definitions that correspond to a given schema description file.

DBGen: This is a bespoke function, generated by the preprocessor when applied to an OCaml-Qanat program, that creates the PostgreSQL database tables from a set of Qanat-OCaml class definitions. It can be called directly by the programmer, or triggered automatically by a configuration file setting when the first session instance is required.

Thus programmers may generate their orm class definitions from an existing database or they may write their own orm class definitions, and generate the database from them. Further, when a program is compiled, the types in the schema description file must match the class definitions, or compile-time fatal type errors will ensue, and, when the program is executed, the user can optionally test the actual database schema against the schema description file to ensure that no changes that could introduce run time type errors have occurred in the database since the program was compiled. In practice, ORMGen generates orm class definitions that are satisfactory for most cases, although some complex cases benefit from manual adjustment.

Transactions ORM frameworks like Hibernate typically use the Spring Framework [JHA+05] to provide an aspect oriented programming solution to transaction management. Since OCaml is a functional programming language, we can take advantage of higher order functions to provide a simpler solution. In Qanat, transactions are functions that take a session factory object and an *operation* function. The operation (also known as a use case function) is a programmer provided function that takes a session and optional arguments. The transaction gets a session from the session factory and invokes the operation with that session. When the operation finishes, the transaction closes the session (automatically persisting any changes back to the database) and commits the database transaction. If an exception is thrown out of the operation, then the transaction rolls back. Using higher order functions in this way relieves programmers from the messy, and error-prone, task of handling the correct object management and exception structures themselves, c.f. Chapter 5.

Matching object and database types At compile time, the schema definition file is read (or generated if it does not exist) and the OCaml code with Qanat orm annotations (i.e. the orm module and orm class definitions) are translated into a bespoke module that provides orm facilities for encapsulated classes. For each orm class in the module, a standard class is generated whose arguments are of the types obtained from the schema definition file and whose fields are of the types obtained from the orm annotations. If the database schema does not match the type of annotated classes, a fatal compile-time type incompatibility error will be triggered, c.f. Chapter 3.

Navigational queries Navigational lazy loading works, as in Hibernate, by providing proxy classes that inherit from persistent classes. Object references within an in-memory persistent object can be references to a (possibly uninitialised) proxy, rather than to the targeted object. When the proxy is first accessed, it triggers a load of the corresponding object from the database, caching the true object reference within the proxy, and forwarding the original access to the now loaded true object. Thus the object graph in memory can be traversed without having to pollute the code with database access calls while still loading into memory only those objects that are actually required, c.f. Chapter 3.

Set oriented queries The Qanat set oriented query language is an SQL like language extension to OCaml that refers to OCaml classes, objects, fields and methods, rather than the database tables, columns and tuples that these elements are mapped to. In particular, object traversal expressions are allowed in the query language and are automatically translated into the appropriate joins, c.f. Chapter 4.

6.2 Programming in Qanat

In this section, we show how to program in Qanat by introducing a simple *hello-world* example and a full-featured *bookshop* example. More examples are included in the Qanat framework distribution.

Our first open source release version of Qanat, together with documentation and example code, is available for downloading from `http://www.cs.bham.ac.uk/~aps/qanat`.

6.2.1 Hello world example

It is traditional for a programming tutorial to start with a "Hello world" example. Here we show how to start Qanat programming by developing a simple "Hello world" program. However, it is not enough to demonstrate Qanat features by simply printing a message to the console, thus, we extend the application with basic orm facilities: save data into, update/delete data and perform queries to retrieve data from, the database.

The project we want to create

We are going to develop a simple message application, in which users can create, save and update messages in the database without writing explicit SQL. The relational table `message` contains a primary-key column that is of type integer, named `id`, and a text column, named `text`, for storing the detail of the message.

id	text
1	"Hello world"
...	...

Note that it does not matter whether such a table already exists in the underlying database or not, as Qanat can create the corresponding table structure based on the orm class definitions, either by setting the property `dbcreate` to `true` in the configuration file, or by invoking the function `init_DB` in a valid session instance.

Qanat adopts the `JSON` format as the configuration and database schema file. Figure 6.1 presents a configuration template, which consists of properties to configure the database connection, and two further properties: `dbcreate` specifies whether the appropriate database structure should be created in the database when the program is executed, and `dbcheck` specifies whether a runtime schema check should be done when the first session instance is initialized from the session factory.

The database schema file is generated by calling the tool `DBSchema` to inspect the database, and is fed into the preprocessor for type checking. Figure 6.2 presents a sample of the schema file extracted, by using `DBSchema`, from database `dbteach`, which contains a table named `message` created for the *helloworld* example. The schema file is saved in the `JSON` format, containing information for all tables existing in the database, table's column information showing each column's *name*, *type* and whether it is *nullable*, is *primary key* or not. The foreign keys and unique constraints are included in the schema file as well.

```
{
  "driver": "database-driver-name",
  "host": "database-host",
  "database": "database-name",
  "user": "db-username",
  "password": "db-password",
  "port": port-number,
  "dbcreate": true-or-false,
  "dbcheck" : true-or-false
}
```

Figure 6.1: Qanat configuration template

```
{
  "db_name": "dbteach",
  "db_tables": [
    { "tbl_name": "message",
      "tbl_columns": [

        { "col_name": "text",
          "col_type": "VARCHAR",
          "col_nullable": false,
          "col_primarykey": false },

        { "col_name": "id",
          "col_type": "INT32",
          "col_nullable": false,
          "col_primarykey": true}
      ],
      "tbl_foreignkeys": [],
      "tbl_unique": []
    }
  ]
}
```

Figure 6.2: Qanat schema file sample

ORM definitions

For simplicity, we define the message class following Qanat ORM conventions, namely orm class and fields are mapped by default to the same name relational table and columns, when the mapping metadata is not specified.

```
module orm MessageApp = struct
   class orm message = object
     val mutable *id : int {auto}
     val mutable text : string
   end
end
```

The compilation of the above code generates a session module, `Session`, inside a `MessageApp` module. The `Session` module provides ORM facilities for the class `message`: mapping the message class to the relational table `message`, and field `id` and `text` to the primary-key column `id` and text column `text` respectively. The property `auto` instructs Qanat to generate and set the value of the primary-key filed `id` automatically when saving message instances into the database.

Use case functions

Based on the orm definition, use cases for saving, updating and querying messages can be defined as follows:

1. Create a new message and save it into the database,
   `save_message: session -> unit`

   ```
   let save_message session =
       let m = new message ~text:"Hello world" () in
       Session.save{message} session m
   ```

2. Load a message according to the given id and modify the text. There is no explicit update/save invoked in code, since Qanat automatically detects and flushes entity modifications into the

database when the transaction is about to close,
`update_message: session -> int -> unit`

```
let update_message session id =
    let m = Session.load{message} session id in
    m#set_text "Hello qanat"
```

3. Query and return all messages containing text "Hello". The execution of the query returns a `(message option) list`, which needs to be filtered out to return a simple `message list`. Of course, if a the list of options is satisfactory, the filtering does not need to be invoked.
`query_message: session -> message list`

```
let query_message session =
    let query = Session.[from message as m where m#text like "%%Hello%%"] in
    Library.optlist_filter (Session.query session query)
```

4. Execute the use case functions inside three separate transactions. The Qanat transaction handler takes a session-factory, initialised from the configuration file, a use case function and an optional argument, as parameters. Function `eval` is used to invoke use case functions without additional arguments beyond the session instance. `txn` invokes use case functions that take a single argument. Both of these transaction functions invoke session flushing when the transaction is about to close.

```
let sf = Session.Factory.default () in
let id = Session.Transaction.eval sf save_message in
Session.Transaction.txn sf update_message id;
Session.Transaction.eval sf query_message
```

5. Execute the three use case functions inside a single transaction.

```
let sf = Session.Factory.default () in
Session.Transaction.txn sf
(fun session id ->
   save_message session;
   update_message session id;
   query_message session) id
```

6. Build and run Qanat program. The complete program is shown in Figure 6.3. The use of CamlP4 as the preprocessor provides the ability to combining the preprocessing and compilation into a single step. That is, the program transformed or generated by the preprocessor is directly fed into the compiler for compilation and linking. The following commands show how to build and run the Qanat program, assuming that Qanat is an installed OCaml package.

```
ocamlfind ocamlc -package qanat -syntax camlp4o -o helloworld helloworld.ml -linkpkg
./helloworld
```

In the case that the schema file is extracted from the database before the compilation, rather than that it will be created in the database by the preprocessor, the schema file can be used for type checking during preprocessing.

```
dbschema -s drivername -h hostname -d databasename -u username -p password
                -port port schemafile.js

ocamlfind ocamlc -package qanat -syntax camlp4o -ppopt "-schema" -ppopt "schemafile.js"
                -o program program.ml -linkpkg
```

```
(* app.js - the configuration file *)

{
    "driver": "pgdriver",
    "host": "localhost",
    "database": "mydatabase",
    "user": "myname",
    "password": "mypassword",
    "port": 5432,
    "dbcreate": true,
    "dbcheck" : true
}

(* helloworld.ml - the program code *)

module orm MessageApp = struct
   class orm message = object
     val mutable *id : int {auto}
     val mutable text : string
   end
end

open Qanat
open MessageApp

let save_message session =
    let m = new message ~text:"Hello world" () in
    Session.save{message} session m;
    Library.valof m#id

let update_message session id =
    let m = Session.load{message} session id in
    m#set_text "Hello qanat"

let query_message session =
    let query = Session.[from message as m where m#text like "%%Hello%%"] in
    let ms = Library.optlist_filter (Session.query session query) in
    List.iter(fun m -> print_endline m#text) ms

let _ =
    let sf = Session.Factory.default () in
    let id = Session.Transaction.eval sf save_message in
    Session.Transaction.txn sf update_message id;
    Session.Transaction.eval sf query_message


(* commands for building and running the example *)

ocamlfind ocamlc -package qanat -syntax camlp4o -o helloworld helloworld.ml -linkpkg
./helloworld
```

Figure 6.3: Qanat helloworld example

6.2.2 Bookshop example

Bookshop is an interactive console application, demonstrating most of features of the Qanat framework. This application is based on a bookshop model, and consists of use cases for daily use of a book shop, including modules and functions from access control, user registration and login, to order placing and checking, and to daily management, etc. These functions are chained together to provide an interactive service, i.e. the client inputs commands, and the terminal returns results.

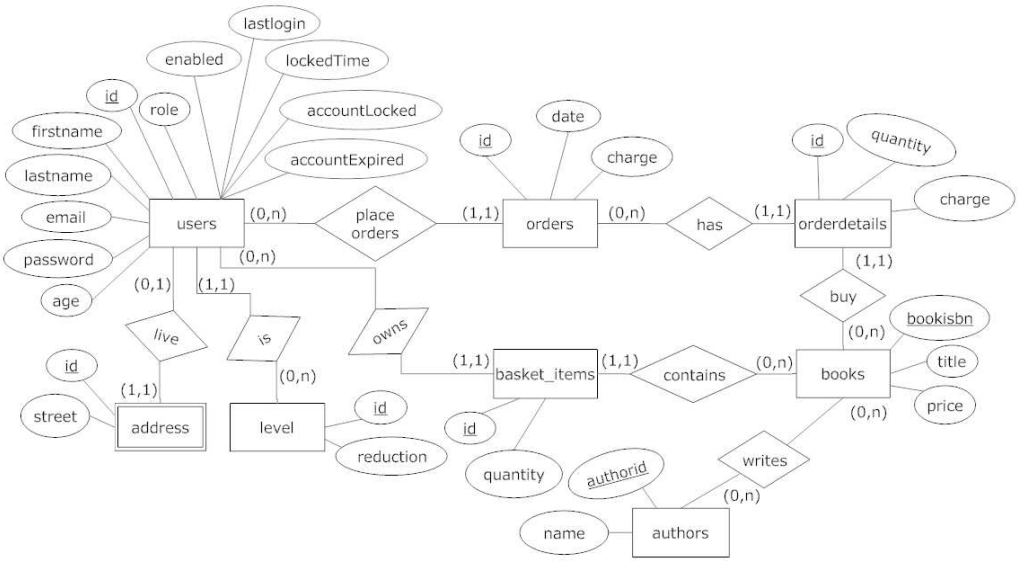


Figure 6.4: Bookshop Entity-Relationship Diagram

Figure 6.4 shows the Entity-Relationship-Attribute (ERA) diagram of the bookshop model. Note that the tuples on the lines connecting entities and relationships should be interpreted as minimum and maximum cardinalities of the participation of the corresponding entity in the corresponding relationship, where a maximum cardinality of n signifies that the cardinality is greater than one. Users in the bookshop system are categorised according to their assigned roles, either customer or administrator. Each customer is associated with a correspondence address, and a reduction level, based on which the charge for an order is calculated. The system records, for each customer, a collection of (completed) orders, each of which has several order details, one for each book in the order, which records the book, quantity and actual price charged. Each customer also has a collection of basket items, identifying a book and the quantity thereof that they are considering buying. On checking out, an order containing these basket items would be created and the basket emptied. The system also maintains the relationship between books and their authors.

The development of the bookshop platform involves various relationships, such as association *many-to-one*, *one-to-many*, *one-to-one* and *many-to-many*, various data types, and shows the use of QQL query, bulk-data-updating, and query composition. For example, orm class `user` is defined with a tuple-type field `name`, unique field `email`, string-type field `password`, variant-type field `role`, record-type field `status`, timestamp-type field `last_login`, and an auto-generated primary-key field `id`. We begin with the declaration of the `user` class, omitting any relationships:

```
class orm user : "users" = object
  val mutable *id : int {auto}
  val mutable name "firstname, lastname" : string * string
  val email : string {unique}
  val mutable password : string

  val mutable role : new role.[Cust when "role".val = "cust"
                              |Admin when "role".val = "admin"]
  val mutable status : new status.{enabled: bool; accountExpired: bool;
                          accountLocked: bool; lockedTime: timestamp option}
    = {enabled= true; accountExpired= false; accountLocked= false; lockedTime= None}
  val mutable last_logintime "lastlogin" : timestamp = Calendar.now ()

  (* overwrite method *)
  method set_password pwd = password <- encrypt pwd

  (* initializer expression *)
  initializer self#set_password password
end
```

Here field `name` maps to a tuple of string columns `firstname` and `lastname` in the `users` table; `email` is defined with a uniqueness property; `role` is of variant type, and has value `Cust` when the discriminator column `role` contains string `cust`, or `Admin` when the discriminator value is `admin`; `status` represents the state of the account, and defined as a tuple with default settings; `last_logintime` records the time of the user's last login, and defined as a timestamp-type field with default time (i.e. the current time). Keyword `new` in the variant and record mapping instructs Qanat to generate the type definition for variant or record during preprocessing. Note that class `user` demonstrates the overwriting of the setter method for field `password` to add an encryption process, and automatically invokes password encryption during object initialisation.

The keyword `new` is extended in Qanat, for use in orm class definitions. The use of '`new`', with record and variants, instructs the Qanat pre-processor to extract the type definition of this record/-variant from the orm mapping definition and generate a standard record/variant type definition. Such a generated record/variant definition is placed before the orm class. That is because, in OCaml, record and variant types must be defined before being used. The extended '`new`' provides an easy way of simultaneously defining and using record/variant types through pre-processing. For instance, the definition of `role` and `status` in the above example would be generated as:

```
type role = Cust | Admin

type status = {enabled: bool;
               accountExpired: bool;
               accountLocked: bool;
               lockedTime: timestamp option}
```

ORM Association in bookshop

Associations play an important role in orm applications for maintaining entity relationships. Qanat supports the definition of these relationships as bidirectional or unidirectional through metadata keywords `key` and `bind` that represent foreign keys in the database):

Note that, to be concise, we omitted basic fields in the orm class definition demonstrated in the following association examples.

1. **one-to-many**, like the relationship between user and placed orders where each user can have many orders but each order was placed by precisely one user. Such a relation is bound through metadata `key` and represented as a collection field of type `resizeSet` or `resizeArray`.

```
meta key_userorder = key {"orders.custid", "users.id"}

class orm user : "users" = object
  val mutable *id : int {auto}
  val mutable orders : order resizeSet with key_userorder on all, delete_orphan {manage}
end
and order : "orders" = object
  val mutable *id : int {auto}
end
```

An ORM associated field in Qanat is defined similarly to basic fields but with additional properties, `key`, `cascade` and `manage`.

- Metadata `key {``t1.a", ``t2.b"}` represents the foreign key constraint from table t1's column `a` to table t2's column `b`.
- Cascade property `all` indicates that all kinds of orm operations, including `save`, `update`, `delete`, will be cascaded down to elements of the collection. `delete_orphan` indicates that orphan elements, i.e. those elements which were in the the collection, but have now been removed from the collection, will be removed from the database before the session is about to close.
- Property `manage` is a newly introduced annotation in Qanat object-relational mapping, used with bidirectional associations, instructing this entity (i.e. user) to maintain the foreign key constraint. Without this property specified on one side of a bidirectional association, both sides, by default, would attempt to update the database when a change to the association occurs.

2. **many-to-one**, such as each order detail records a book bought by the user. Such a relation is expressed as a singleton or optional entity field based on the whether the foreign key constraint allows nullable value.

   Note that there is no need to specify the property `manage` in this case, since, in this unidirectional association, the foreign key constraint is stored in the current entity (i.e. `order_detail`), the update of `order_detail` automatically (by default) updates the relationship as well.

```
meta key_detailbook = key {"orderdetails.bookisbn", "books.bookisbn"}

class orm order_detail : "orderdetails" = object
  val mutable *id : int {auto}
  val mutable book : book with key_detailbook on save_update
end
and book : "books" = object
  val mutable *isbn "bookisbn" : string {assign}
end
```

3. **many-to-many**, for example, the relationship between books and authors. In the database, such a relation is stored in a third, ancilliary table, via two foreign key constraints. Qanat provides the structure *bind*, representing the combination of two keys that are used as the many-to-many foreign key constraints.

   In this example, we build a bidirectional relationship. We represent the authors using an array structure that keeps the ordering of authors. Two entities are updated in order to achieve a change in this relationship. Qanat will detect both changes when the session is flushed, and each change will trigger an update to the same relationship. This results in two separate updates that set the relationship to the same value. In order to avoid the unnecessary second update, we must select one side of the relationship to manage such changes so that only changes to the entity on that side will trigger an update to the relationship. We make this selection by adding the `manage` property to the relationship field on the selected side of the relationship.

```
meta key_auth = key {"authorbooks.authorid", "authors.authorid"}
meta key_book = key {"authorbooks.bookisbn", "books.bookisbn"}

meta bind_authbook = bind {key_auth, key_book}

class orm book : "books" = object
  val mutable *isbn "bookisbn" : string {assign}
  val mutable authors : author resizeArray {idx: "author_idx"}
                        with bind_authbook on save_update {manage}
end
and author : "authors" = object
  val mutable *id "authorid" : int {auto}
  val mutable books : book resizeSet with bind_authbook on save_update
end
```

4. **one-to-one**, for example, each user is registered with at most one correspondence address. The address entity is a weak entity depending on the user entity. That is, the removal of user record invokes the removal of the bound address as well and the address record shares the same primary key with the corresponding user.

   To express such a relation, the `address` class is designed with a `refer_to` primary-key strategy, i.e. the value of the primary-key field depends on that of an associated orm field. The cascade property `all` on field `address` in class `user`, implies that operations including `delete` performed on the user entity will be cascaded to the associated address.

```
meta key_addrusr = key {"address.id", "users.id"}

class orm user : "users" = object
  val mutable *id : int {auto}
  val mutable address : address option with key_addrusr on all
end
and address : "address" = object
  val mutable *id "id": int {refer_to user}
  val mutable user : user with key_addrusr on save_update
end
```

QQL in bookshop

QQL queries are embedded in the program, using a syntax similar to SQL, but based on object-oriented classes and fields and supports querying in a navigational way (i.e. following object references) as well as in a set-oriented way. Queries in QQL are written in the form `Session.[query-expression]`, where `Session` is the orm session module, and executed through the query interface,

```
Session.[select * from class where expr] (* query expression *)
Session.query session query (* execute query *)
Session.exec_update session query (* execute bulk-data-update *)
```

The current version of QQL is case sensitive, but supports query keywords in a complete lowercase or uppercase, but not mixed case form, e.g. `select` or `SELECT`. Considering the example of the bookshop application, we give a description of QQL queries.

1. **The from clause**

   The `from` clause in QQL queries is compulsory, while the other clauses are optional. The simplest QQL query is of the from:

```
from user
```

which returns all users from the session instance, in the form of `(user option) list`.

Like SQL, QQL allows assigning aliases for querying classes, and supports cartesian product over multiple classes, such as

```
from user as u, order as o
```

for returning a list of user and order pairs, `(user option * order option) list`.

Note, that QQL returns entities as option type based on the fact that join operations like *left-join*, and *right-join* may result in empty columns, and that some columns in the database may contain nulls.

2. **Associations and joins**

   Unlike SQL, QQL allows writing queries in a navigational way. We believe that, by following object references, queries tend to be shorter and more intuitive. For example,

   ```
   select u#name, u#level#reduction from user as u
   ```

   returns the charge reduction value of all users. Note that this involves an implicit join. QQL supports object associations in most clauses except `limit` and `offset`.

   Explicit join, including the *left*, *right* and *inner* join, can be defined with a join condition.

   ```
   select user, address
   from user left join address on user#id = address#id
   ```

   The above query joins class `user` and class `address` based on the equality of identifiers of user and address.

3. **The select clause**

   The `select` clause in QQL is optional; that is, when `select` is not specified, Qanat will return all entities in the query result.

   ```
   from user
   ≡ select * from user
   ≡ select u from user as u
   ```

   The `select` clause allows returning entities, collection of entities, entity properties that can be of any type including component types (e.g. record, tuple), as well as constant values and aggregate expressions. For example, the following query returns a list of pairs, where the first element of each pair is a user's email address and the second element is a collection of that user's placed orders.

   ```
   select u#email as email, u#orders as orders from user as u where u#role = Cust
   ```

   The alias of each returned item can be referred to in the order-by clause or when the query is used as a sub-query in query composition.

   The optional `distinct` keyword requires duplicated values to be omitted from the query result.

4. **Aggregate expressions**

   QQL supports two kinds of aggregate expressions, one is based on the group-by clause, and the other one is based on the collection of object fields.

   The group-by aggregation is similar to SQL's aggregation (omitting the `group by` clause means that, as in SQL, the aggregation function is calculated over all the data matching the query conditions, rather than separately for each group of data. For example, a query for returning the count of users and the average age can be defined as,

```
select count(*), avg(u#age) from user as u
```

The collection based aggregation is invoked as collection methods and are evaluated over the corresponding collection object, such as

```
select u#email, u#orders#count(), u#orders#[x => x#charge]#sum() from user as u
```

where `orders#[x => x#charge]` means to map each `order` element in the `orders` collection of `u` to a collection of the `charge` field values of those elements before calculating the `sum` over those charge values. Note that, although this was described in terms of OCaml objects and values, the query will be translated into SQL and executed by the database server, and not in memory in OCaml code.

QQL supported aggregate functions include `count`, `sum`, `avg`, `max` and `min`. Where `sum` and `avg` can be applied on numeric values, `max` and `min` on comparable values, and `count` is suitable for all kinds of values.

5. **The where clause**

   The where clause defines a boolean predicate, used to reduce the number of rows returned by the query. That is, the query result consists of rows on which this predicate has evaluated to `true`.

   As an example, we define variable `v` in the following query, trying to return all users who have placed orders, the total charge of these orders is over a specified amount.

```
from user as u where u#orders#[x => x#charge]#sum() > :v
```

6. **Variables, components and expressions**

   Variables in QQL are declared as, : v, where `v` is the variable name and the type of which is inferred in a standard way during the compile time. Except clauses like `select`, `group-by` and `order-by`, variables can be used in any other clauses. We discuss more in query composition when using variables in the `from` clause.

   The following example demonstrates using variables in where and limit clauses:

```
(* return all orders placed at the current day, and limit the results to 10 rows *)
let q = Session.[from order where order#date = :now limit :n] in
(* execute query *)
Session.query sess (q ~now:(Calendar.Date.today ()) ~n:10)
```

   In addition to primitive types like integer, float, date and numeric, etc, QQL supports using and returning composite types in queries, including the type of *option*, *tuple*, *record*, *object*, *variant* and *collection*.

   Suppose class `user` is defined with fields `role`, `name`, `status`, `address` and `age`, an example query can be defined as,

```
select u#email, u#orders#[x => x#charge]#max() from user as u
where u#role = Cust
      and u#name.{0} like :firstname
      and u#status.enabled = true
      and u#address#street in :street_list
      and (valof u#age) between :min_age and :max_age
```

   where field `role` is of type variant that consists of constructor `Admin` and `Cust`; `name.{0}` returns the first element from the tuple `name`; `status.enabled` denotes the access of field values from record `status`; function `valof` extracts value from an option-type value.

QQL queries support most expressions that are used in SQL, including logical (`and`/`or`/`not`), arithmetic (+, −, ×, /) and comparison (>, <, =, <>) operations, as well as `between-and`, `in`, `like`, `is-null`. Since the host language OCaml lacks support for function overloading, QQL defines a series of arithmetic operators for various types, such as `+` for integer addition, and `+.` for floating point number addition.

7. **The group-by, having clause**

   The `group-by` clause defines criteria for the partition of query result sets into groups, used for the calculation of aggregate expressions. It differs from SQL in that `group-by` in QQL requires providing an alias for each path expression involved. These aliases can then be referred to in clauses like `select` and `order-by`. The `having` clause specifies a boolean predicate for filtering out these groups; group-by aggregate expressions can be used in the having clause.

   When using `group-by`, queries can only return values mentioned in the `group-by` clause, in addition to aggregate values. However, because of the support for navigational queries, it becomes possible to return any value that is associated with the keys (the `group-by` criteria) of the `group-by` clause.

   For example, here is a query for returning user email, the charge reduction of the associated user level, and average order charge for users who have placed orders with a total charge over a specified amount.

```
select k#email, k#level#reduction, avg(order#charge)
from user join order on user#id = order#owner#id
group by user as k
having sum(order#charge) > :v
```

   Alternatively, an equivalent query can be defined by using collection-based aggregation, and seems to be more intuitive,

```
select u#email, u#level#reduction, u#orders#[x => x#charge]#avg()
from user as u
where u#orders#[x => x#charge]#sum() > :v
```

8. **The order-by, limit and offset clause**

   Clauses like `order-by`, `limit` and `offset` remain the same as in SQL. The `order-by` clause identities the ordering criteria, based on which fields and directions the query results should be sorted in. Path expressions used in the `order-by` clause can be constructed from query sources defined in the `from` clause, or aliases specified in the `group-by` and `select` clause. `limit` and `offset` help customise the return of the query; both of these two clauses accept an integer value that can be constant value or a variable.

   For example, the following query returns the last 10 login users,

```
from user as u
order by u#last_logintime desc
limit 10
```

   The ordering of query results is never guaranteed unless an `order-by` clause is specified.

9. **Query composition**

   Rather than concatenating queries as strings, QQL supports composing queries as components in a compile-time type-safe way. That is, queries are defined as objects that can be passed as variables to another query and used as query sources in a way similar to orm classes.

   The simplest, but trivial, example of query composition is

```
from :q
```

which defines a query-type variable `q` and returns it directly without additional operations.

In the following example, we define a few queries, then compose these queries to get a new use case query.

```
(* a generic paging query, which sets limit and offset for the argument sub-query *)
let page = Session.[from :q limit 10 offset :n]

(* return all users registered with a specified email domain address *)
let q_users = Session.[from user where user#email like :domain]

(* return user's email and the total charge of placed orders from the argument sub-query *)
let q_sums = Session.[
      select q#user#email as email, q#user#orders#[x => x#charge]#sum() as sum
      from :q
      order by email asc, sum desc]

(* construct a new query by composing the above queries:
   Find the email address and total charge of their placed orders of users whose
   email address ends in "@qanat.com", and return 10 records starting
   at the 20th.
*)
let query = page ~q:(q_sums ~q:(q_users ~domain:"%@qanat.com")) ~n:20
```

10. **Bulk data updating**

    As an alternative to orm operations that require loading all related data for updating, Qanat provides an efficient approach of bulk updating of data via QQL `update` and `delete` statements.

    Unlike read-only query statements, each updating query (including `update` and `delete`) can only act on a single entity class type. The optional `where` clause specifies which instances (corresponding to rows in the database) the action would operate on. The execution of these queries returns an integer value indicating the number of affected rows by the current query.

    Consider deactivating accounts of users who have not logged in within a specified time. Such an operation performed without bulk data updating would require loading all user instances and then iterate these instances to perform checking and updating. With bulk data updating, it can be expressed simply, and efficiently executed, as:

    ```
    update user as u set u#status.accountExpired = true
    where :now --! u#last_logintime > :expire_period

    (* (--!): Calendar.t -> Calendar.t -> Calendar.Period.t *)
    ```

    where operator `--!` calculates the period between two given calendar times.

Use case in bookshop

Here we implement a number of bookshop daily use cases, demonstrating the basic ORM features of the Qanat framework. Because of the proxy-based lazy loading and automatic modifications flushing, there is no need to explicitly load related data from and write modifications back into the database. The transaction handler invokes the use case function inside a database transaction, and commits the transaction when the use case finishes, or rolls back when any exception is thrown.

Figure 6.5 is a function defining an operation to make an order from a basket of items. The use case carries out the following steps:

1. Identify customer by email, if the customer cannot be found, raise an exception
2. Create order_details and record the actual charge for each of the books in the customer's basket
3. Clear customer's basket

4. Create the order and calculate the total charge according to the reduction level
5. Save all of these order_details and the order to the database

```
exception NonExistUser of string

let make_order sess email =
  let user =
      let q = Session.[from user where user#role = Cust and user#email = :email] in
      try Session.query sess (q ~email) with
      |[Some user] -> user
      |_ -> raise (NonExistUser email)
  in
  let order = new order ~charge:Numeric.zero () in
  let total_charge =
      ResizeSet.fold(fun b acc ->
        let charge = Numeric.mult b#book#price (Numeric.of_int b#quantity) in
        let detail =
          new order_detail ~quantity:b#quantity ~book:b#book ~charge:charge ()
        in
        ResizeSet.add detail order#orderdetails;
        ResizeSet.remove b user#basket;
        Numeric.add charge acc
      ) user#basket Numeric.zero
  in
  order#set_charge (Numeric.mult total_charge user#level#reduction);
  ResizeSet.add order user#orders
```

Figure 6.5: Bookshop usecase: Making order

The function in Figure 6.6 defines an operation to find and print the best-selling books as evidenced by the ordering history. Note that, unlike Hibernate, Qanat is able to automatically wrap the result set into an appropriately typed list, namely *(book option * int) list*, without explicit type casting. Also note that for currency, we are using a Numeric type. This type Qanat's extended version of OCaml's Num type, a large rational number type suitable for handling decimal values without roundoff errors.

Figure 6.7 demonstrates how to execute an operation, in this case the `make_order` operation, in a transaction. The session object will be closed at the end of the transaction, and all changes persisted to the database at that point. If the operation throws an exception, the transaction will automatically be rolled back. Otherwise it will be committed.

```
let bestselling_books sess n =
  let query = Session.[
       SELECT book, sum(detail#quantity) as cnt
       FROM order_detail as detail
       GROUP BY detail#book as book
       ORDER BY cnt DESC
       LIMIT :n ]
  in
  let book_counts = Session.query sess (query ~n) in
  List.iter(fun (bookopt, cnt) ->
     match bookopt with
    |None -> ()
    |Some book -> Printf.printf "ISBN: %s Title: %s Count: %d\n"
                        book#bookisbn book#title cnt
  ) book_counts
```

Figure 6.6: Bookshop use case: Find best-selling books

```
let _ = let sf = Session.Factory.default () in
        Session.Transaction.txn sf make_order email
```

Figure 6.7: Execute use case in a transaction

6.3 Qanat grammar

Qanat includes language extensions for the embedding of object-relational mapping (ORM) and query language (QQL) in standard OCaml. One of our aims was to minimize modifications to the host language. This section presents the formal grammar of these extensions.

6.3.1 Meta grammar

Qanat grammars are defined by using a meta grammar as follows,

::= defines a **grammar entry**, which consists of a number of grammar rules.

| separates **alternative rules**.

< > denotes **standard OCaml grammars**.

( ) defines the **scope** of a rule element.

[ ] specifies an optional rule element.

$[\ ]^+$ denotes that there is one or more of the specified element.

$[\ ]^*$ denotes that there are zero or more of the specified element.

TERMINAL uppercase and italic font, denotes **terminal entries** defined as lexer tokens in section 6.3.4.

non-terminal lowercase and italic font, denotes **non-terminal entries**.

6.3.2 ORM grammar

Qanat's object-relational mapping extension enable embedding the mapping metadata directly in the program code, and is based on the standard OCaml language.

Let *id* and *uid* represent valid identifiers with an initial lowercase or uppercase letter respectively, primitive entries for naming are defined as:

```
moduleName            ::= uid
keyName, bindName     ::= id
ormName, fieldName    ::= id
tableName, columnName ::= id
funcName              ::= id
```

where module name is an initial uppercase letter identifier and other names are initial lowercase letter identifiers.

Core Orm

The syntactic extension for ORM facility introduces two extended structures: an orm module and an orm class, as well as a key and bind declaration. The keyword 'orm' on the definition indicates that this structure will be preprocessed to incorporate an ORM facility; 'meta' denotes that this value will be used during and removed after the preprocessing.

```
module-expr ::=
              <standard OCaml module expression>
            | META keyName EQ key-def
            | META bindName EQ bind-def
            | CLASS ORM orm-def [AND orm-def]*
            | MODULE ORM moduleName EQ STRUCT module-expr END
```

The orm module is transformed into a module of the same name, and includes a module named `Session`, which provides ORM facilities including `save`, `update`, `delete` and `load` for each managed orm class based on the class-embedded mapping metadata; the preprocessing of each orm class results in a standard class that is managed by the session module, and obtained by the removal of the mapping metadata.

```
key-def ::=
        KEY LBRACE LDQUOTE tableName DOT columnName RDQUOTE COMMA
          LDQUOTE tableName DOT columnName RDQUOTE RBRACE

bind-def::= BIND LBRACE keyName COMMA keyName RBRACE
```

Metadata **key** represents a database foreign key, specifying a foreign key constraint from a column in one table to the primary key of another table. Metadata **bind** consists of two keys, representing the *many-to-many* relationship between two relational tables. Such a relationship requires an additional table to store the relation.

```
orm-def ::=
        ormName [COLON LDQUOTE tableName RDQUOTE] EQ OBJECT
           orm-str-item [orm-str-item]*
        END

orm-str-item ::=
              field-expr
            | <standard OCaml class method definition>
            | <standard OCaml class initializer expression>
```

The ORM class (c.f. entry *orm-def*) defines an OCaml class (named *ormName*), the embedded mapping information denotes that this class is mapped to a relational table, *tableName*. Note, that *TableName* is optional. When it is not explicitly specified, the class is mapped to a table of the same name as the class.

As in a standard class definition, an orm class consists of a number of class structure items, including field expressions, method definitions, and class initializer expression (c.f. entry *orm-str-item*).

Depending on the mutability of each field, corresponding `getter` and `setter` method will be generated, the `getter` is named with the same name as the field, the `setter` is named as `set_fieldname`. The user-defined methods and class initializer expressions are left unchanged in the translated class.

field-expr ::=

(1) *VAL* [*MUTABLE*] *STAR fieldName* [*LDQUOTE* column-mapping *RDQUOTE*] *COLON primitive-type* [*STAR primitive-type*]* *LBRACE key-generator RBRACE*

(2) | *VAL* [*MUTABLE*] *fieldName* [*LDQUOTE* column-mapping *RDQUOTE*] *COLON field-type* [*LBRACE UNIQUE RBRACE*] [*EQ expr-val*]

(3) | *VAL* [*MUTABLE*] *fieldName COLON assoc-type WITH* (*keyName* | *bindName*) [*ON cascade-def*] [*LBRACE MANAGE RBRACE*]

Three kinds of field expressions can be used in an orm class, c.f. *field-expr*: a primary-key field, a plain field and an associated field.

1. Primary-key field, identifies primary-key columns in the relational table. It differs from other fields in that that it is defined with an asterisk '*', c.f. the first entry of *field-expr*. Each orm class must have precisely one primary-key field, and that must be of basic type (*primitive-type*), or a tuple type, constructed from basic types, when the mapped table has more than one primary-key column. To support this, the primary-key field in Qanat may contain multiple columns in the form of tuple.

   primitive-type ::=
   INT | *INT32* | *INT64* | *FLOAT* | *CHAR* | *STRING* | *BOOL*
   | *NUMERIC* | *NUMERIC(int)* | *NUMERIC(int, int)*
   | *DATE* | *TIMETZ* | *TIMETZ* | *TIMESTAMP* | *TIMESTAMPTZ*

   key-generator ::=
   AUTO
   | *ASSIGN*
   | *REFER_TO fieldName* [*COMMA fieldName*]*

   Primary-key fields are defined with a property, **key generator**, indicating how to obtain primary key values when this class instance is saved into the database.

   (a) *auto* – the value of the primary-key will be automatically generated when the instance is persisted into the database. For primary-key fields with numerical types (e.g. int, int32, etc), the value will be generated from an increasing sequence; for those with calendar types, the current date or time will be used as the primary-key value.

   (b) *assign* – users take the responsibility of assigning the primary key value during entity initialization.

   (c) *refer_to* – the value of primary-key will be obtained from other fields, namely the specified *fieldName* following keyword `refer_to`. Note that this requires these referred fields to be associated fields in the current class.

2. Plain field, corresponds to ordinary columns in the relational table, having a wider range of types including: *option type*, *tuple*, *record*, *variant* and *anonymous object*, see the second entry of *field-expr*. These types, except the *option type*, are called complex types and used to map multiple columns into a single field.

   The mapping of class fields (including primary-key and plain fields) into relational column is specified through the optional property, *column-mapping*. When a column-mapping is not defined explicitly, the field is mapped to a same name column. Unlike primary-key fields, plain fields can be defined with a default value.

   column-mapping ::=
   column-mapping [*COMMA column-mapping*]*

```
| LBRACKET column-mapping RBRACKET
| columnName
| UNDERSCORE
```

Column-mapping defines the mapping between class fields and table columns.

(a) When a field is mapped to a single column, the mapping is denoted by an identifier, *columnName*. c.f. the third entry of *column-mapping*.

(b) In the case of a tuple type, the mapping is denoted as *columnName* [*COMMA columnName*]*. *LBRACKET* and *RBRACKET* are optionally used depending on whether brackets appear in the tuple definition.

(c) For fields of complex types (excluding tuple type), the *column-mapping* can be omitted or denoted as a place holder, *UNDERSCOPE*. The real column mapping is specified in the type definition. Further discussion is shown in the following part.

3. Associated field, represents the relationship between orm entities. This kind of field is defined with a specified key or bind, representing the corresponding foreign key constraints, c.f. the third entry of *field-expr*. Associated fields can be of type orm-class, optional orm-class, or orm-class based collection (*resizeSet* or *resizeArray*), c.f. *assoc-type*.

```
assoc-type ::=
     ormName
   | ormName OPTION
   | ormName RESIZESET [ LBRACE CMP COLON compareFunctionName RBRACE]
   | ormName RESIZEARRAY LBRACE IDX COLON LDQUOTE columnName RDQUOTE RBRACE
```

ResizeSet has an optional property, **cmp**, which specifies the comparison function used in the collection. When *cmp* is not specified, resizeSet uses the default comparison, namely comparing two orm entities by the value of primary-key fields when both of them are in a persistent state or have assigned primary keys, otherwise comparing the physical memory address directly.

ResizeArray requires an index, numerical-type column, used to represent the order of elements.

```
cascade-def ::=
              cascade-def [COMMA cascade-def]*
            | ALL
            | DELETE
            | SAVE_UPDATE
            | DELETE_ORPHAN
```

Associated fields have optional properties, *cascade* and *manage*: *cascade* indicates whether orm operations on the current class instance should be cascaded to this associated field. The *manage* property only takes effect in bidirectional relationships, indicating which entity maintains the relationship.

The cascade property can be defined by using the following tags:

- *ALL* - all orm operations, including save, update, delete and delete_orphan, would be cascaded to this associated field.
- *DELETE* - delete action would be performed in a cascade manner. It means that when this entity is deleted, all of its directly associated entities would be deleted from database as well.
- *SAVE_UPDATE* - save and update operation would be cascaded to associated entities.
- *DELETE_ORPHAN* - orphan elements (i.e. those elements that once existed in the collection but were removed later) would be deleted from the database. This property can only be applied to collection-type fields.

More on Data types

In addition to primitive types, Qanat supports the use of complex, even nested, data structures as field type, including *tuple*, *record*, *variant* and *anonymous object*. These kinds of types require the ability to map multiple table columns into a single field.

```
    field-type ::=
(1)       field-type [STAR field-type]+
(2)     | record-type
(3)     | variant-type
(4)     | NEW record-type
(5)     | NEW variant-type
(6)     | object-type
(7)     | primitive-type OPTION
(8)     | primitive-type
(9)     | LBRACKET field-type  RBRACKET
```

Entry 1 of *field-type* represents a tuple type, inside which each single type is separated by an asterisk. Entries 2 to 7 represent *record*, *variant*, *anonymous object* and *optional primitive type*, respectively.

Types *record* and *variant* in OCaml must be defined before being used. However, Qanat requires that the declaration of these types be repeated on the fields, rather that just using the type name alone. This is so that the user can specify the column names that are used in the mapping. This is discussed in more detail below. Repeating the declaration for the types is inconvenient to the programmer, so Qanat provides support in the form of placing the keyword *new* before the record and variant mapping, requesting the extraction and pre-generation of the type definition of record or variant, c.f. entry 4 and 5 of *field-type*. Thus, while the declaration of the *record* and *variant* types must still appear in the field itself, the programmer does not need to explicitly declare the type on its own before the declaration of the field.

```
record-type ::=
      id DOT LBRACE record-item [SEMICOLON record-item]*  RBRACE

record-item ::=
      [MUTABLE] fieldName [LDQUOTE column-mapping RDQUOTE] COLON field-type
```

To embed the mapping information, the use of record (or variant) requires declaring the name and internal structure of the record (or variant) simultaneously, denoted that the record (or variant) name is followed by the internal structure, inside which the mapping information is embedded. This internal structure is the same as appear in the declaration of standard records, i.e. that record fields can be either mutable or immutable, and separated by semicolons. Since record field can be of arbitrary type (see *field-type*) this means that nested records are allowed. As an extension, each field inside the record is defined with an optional column-mapping that specifies the corresponding table column in the database; when the type of the field is a complex type, this column-mapping is omitted or replaced by a underscore (place holder), implying the exact relational mapping is specified inside the type definition in a way similar to the use of record.

```
variant-type ::=
   id DOT LSQRBRACKET variant-item [VERTICALBAR variant-item]*  RSQRBRACKET

variant-item ::=
   [']uid [LDQUOTE column-mapping RDQUOTE] [OF field-type]
      WHEN LDQUOTE columnName RDQUOTE DOT VAL  EQ expr-value
```

A variant type is used in the case that a discriminator column distinguishes row data in a relational table. Each constructor (denoted as *uid*) in the variant corresponds to a case when the discriminator column has a specified value. The expression `(when ‘‘col’’.val = expr)` specifies the value of the discriminator column. Note that constructors could have multiple (or zero) parameters (term `(of field-type)`) used to represent the values of columns that are bound to the discriminator column. Optional ['] placed before the constructor implies creating a polymorphic variant type [LDG+07], instead of a standard variant. Polymorphic variants are used to remove the assumption that every constructor in OCaml reserves a name to be used with a unique type. Without polymorphic variants, one could not use the same constructor name in another type.

object-type ::=
  LANGLE obj-str-item [*obj-str-item*]* *RANGLE*

obj-str-item ::=
    obj-field-expr
  | <standard OCaml class method definition>
  | <standard OCaml class initializer expression>

obj-field-expr ::=
  [*MUTABLE*] *fieldName* [*LDQUOTE* column-mapping *RDQUOTE*] *COLON field-type*
    [*EQ expr-value*]

Anonymous object types are quite similar to orm classes, consisting of fields, methods and initializer expression, except that they do not have an explicit type abbreviation (i.e. class name). Since anonymous classes are used to group a number of columns rather than a relational table, primary-key fields can not be declared in anonymous objects.

Qanat Expressions

The Qanat language extension includes syntactic extensions for invoking orm functions and embedding queries in the program.

expression ::=
    <standard OCaml expression>
  | [*moduleName DOT*]* *orm-function LBRACE ormName RBRACE*
  | [*moduleName DOT*]+ *LSQRBRACKET expr RSQRBRACKET*

orm-function ::=
    LOAD | *SAVE* | *UPDATE* | *SAVE_UPDATE* | *DELETE* | *INIT*

ORM functions, including *load*, *save*, *update*, *save_update*, *delete*, and *init*, are pre-generated for each orm class in the session module. Note that function `init` is used to initialized proxies existing in a specified entity. The invocation of these functions are expressed in a uniform manner:

```
MySession.funcname{ormclass} session entity_instance
```

where *funcname* denotes the function name, and *ormclass* is the type of the instance on which the orm function is defined.

Queries in Qanat are written as standard expressions in the form of,

```
let query = MySession.[select-from-where]
```

In section 6.3.3, we present the formal grammar of Qanat queries.

Session Module Interface

The Qanat session module provides ORM and query functionality for managed orm classes. The interface of the session module mainly includes functions for orm class persistence (e.g. `save`, `load`, etc), database construction, session inspection, query execution, and sub-modules for session factory and transaction handling.

1. ORM Related Functions

```
type session

(* database table generation or drop functions *)
val init_DB : session -> unit
val drop_DB : session -> unit

val gen_tables : ~driver:string -> ~host:string -> ~database:string
          -> ~user:string -> ~password:string -> ~port:int -> unit

(* session inspection functions *)
val is_open : session -> bool
val flush : session -> unit
val close : session -> unit
val statistics : session -> unit

(* orm functionality for each of orm types *)
type ormclass (* orm class type *)
type idtype (* type of primary key field *)

val load{ormclass} : session -> idtype -> ormclass
val save{ormclass} : session -> ormclass -> unit
val update{ormclass} : session -> ormclass -> unit
val save_update{ormclass} : session -> ormclass -> unit
val delete{ormclass} : session -> ormclass -> unit
val init{ormclass} : session -> ormclass -> unit

(* query type, either queryonly or updatable *)
type queryonly
type queryupdate
type ('queryType, 'returnType) query

(* query execution functions *)
val query : session -> (queryonly, 'rtype) query -> 'rtype
val exec_update : session -> (queryupdate, int) query -> int
val show_sql : session -> ('qtype, 'rtype) query -> string
```

2. Factory Functions (SessionModule.Factory)

```
type sf

(* initialize factory from default config file, app.js *)
val default : unit -> sf

(* initialize factory from config data *)
val make : Config.t -> sf

(* initialize factory by JSON format config file *)
val make_fromFile : string -> sf
```

3. Transaction Functions (SessionModule.Transaction)

```
(* evaluate functions without additional parameters(excepts the one for session) *)
val eval : Factory.sf -> (session -> 'rtn) -> 'rtn

(* evaluate functions without one additional parameter(excepts the one for session) *)
val txn : Factory.sf -> (session -> 'arg -> 'rtn) -> 'arg -> 'rtn

(* for multiple session-factories, see example for more information *)
type ('a, 'b, 'c, 'd) t
```

```
val txnK : Factory.sf -> (session -> 'a -> 'b) -> ('a, 'b, 'c, 'd) t
val txnN : Factory.sf -> (session, ('a -> 'b), 'c, 'd) t -> ('a, 'b, 'c, 'd) t
val txnExc : ('a, 'b, 'a, 'b) t -> 'a -> 'b
```

6.3.3 QQL grammar

The current version of QQL is case sensitive: query keywords are expressed either in lowercase or uppercase, but not mixed case. For example, **from** could be written as **from** or **FROM**, but words like '**From**' is not recognized.

qanat-query ::=
[*select-clause*] *from-clause* [*where-clause*] [*groupby-clause*]
[*having-clause*] [*orderby-clause*] [*limit-clause*] [*offset-clause*]
| *UPDATE className SET set-expr* [*COMMA set-expr*]* [*where-clause*]
| *DELETE FROM className* [*where-clause*]

Object-oriented Querying

A QQL query statement consists of clauses, including *select*, *from*, *where*, *group-by*, *having*, *order-by*, *limit* and *offset*. The *from* clause is compulsory, while the others are optional. When *select* is not specified, Qanat infers the return result from the *from* clause automatically, returning all orm instances identified by the query.

from-clause ::= *FROM from-expr* [*COMMA from-expr*]*

from-expr ::= (*path* | *variable*) [*AS id*] [*join-expr* (*path* | *variable*) [*AS id*] *ON join-cond*]*

join-expr ::= [*LEFT* | *RIGHT* | *INNER*] *JOIN*
join-cond ::= *id SHARP path EQ id SHARP path*

variable ::= *COLON id*

The *from* clause defines query sources that can be orm classes (expressed as *path*) or query-type variables. Expressions of the form *from* e_1, e_2 represents a cartesian production over e_1 and e_2. Explict joins can be defined in a manner that is similar to SQL but based on object-oriented objects and fields.

Variables in QQL are denoted as a colon followed by the variable name Based on the query context, the type of the variable is inferred in a standard way and the value is bound to the query as a labelled variable.

path ::=
(1) *path SHARP id*
(2) | *path DOT id*
(3) | *path DOT LBRACE int RBRACE*
(4) | *AMPERSAND path*
(5) | *VALOF path*
(6) | *path DOLLAR*
(7) | *ITEMSOF path*
(8) | *path SHARP LSQRBRACKET id RIGHTARR expr RSQRBRACKET*
(9) | *path SHARP funcName* ()
(10) | *LBRACKET path RBRACKET*
(11) | *id*

The *path* expression represents the return of values from various objects: the **1st** entry for accessing a field value from class objects, `obj#v`; the **2nd** for returning value from a record, `r.field`; **3rd** for getting the nth element from a tuple value, `t.{n}`; the **4th** and **5th** entries extract the value from an optional value, `valof v`; the **6th** and **7th** entries getting an element from a collection,

used for type conversion from collection type to a single type, `itemsof collection`; the **8th** entry representing the mapping of a collection for returning a new collection, `col#[x => e]`; the **9th** entry invoking aggregate functions over a collection, `col#f ()`; the **10th** wrapping path in a pair of brackets, `(p)`; the last entry denoting a lowercase identifier.

select-clause ::=
SELECT [*DISTINCT*] *select-property* [*COMMA select-property*]*

select-property ::=
(*STAR* | *constant* | *path* | *aggr-expr*) [*AS id*]

The *select* clause specifies the returned values of a query, consisting of *constant* values, *path* expressions and *aggregate* expressions. Each of these items is separated by a comma. Optional *DISTINCT* filters out duplicate results; *STAR* represents returning all entities matched by the query. Note that the select clause supports defining aliases for each returned value. These names can be used for returning corresponding query results in the query composition.

constant ::= *int* | *float* | *char* | *string* | *bool*

aggr-expr ::=
COUNT (*STAR* | *path*)
| *other-aggr-function path*

other-aggr-function ::= *SUM* | *AVG* | *MAX* | *MIN*

QQL supports aggregate functions, including *count*, *avg*, *max*, *min* and *sum*. As in SQL, *count* allows the use of *STAR* to represent all the return values.

The results of QQL queries are returned as explicitly typed data. Since (left, right and outer) join operations may result in null values, and some values may be null in the database anyway, *path* expressions are wrapped as option types. Other kinds of *select* items are returned as their original type.

where-clause ::= *WHERE expr*

expr ::=
NOT expr
| *expr AND expr*
| *expr OR expr*
| *expr IS* [*NOT*] *NULL*
| *expr IN variable*
| *expr IN LSQRBRACKET expr* [*SEMICOLON expr*]* *RSQRBRACKET*
| *expr BETWEEN expr AND expr*
| *expr LIKE expr*
| *expr infix-cmp expr*
| *expr infix-op expr*
| *path*
| *variable*
| *constant*

The *where* clause specifies a boolean predicate (*expr*), used to restrict the number of rows returned by the query. The rows on which the boolean predicate does not evaluate to *true* are eliminated from the query results. The *where* predicate is defined through a boolean expression consisting of *variable*s, *path* expressions and *constants* as well as logic, comparison and arithmetic operations (e.g. *and*, *or*, *in*, *between-and*, *is-null*, *greater-than*, *plus*, etc).

```
infix-cmp ::= EQ | NEQ | GT | GE | LT | LE

infix-op  ::=
            STRINGPLUS | PLUS | MINUS | MULT | DIV
          | FLOATPLUS | FLOATMINUS | FLOATMULT | FLOATDIV
          | NUMPLUS | NUMMINUS | NUMMULT | NUMDIV
          | INT32PLUS | INT32MINUS | INT32MULT | INT32DIV
          | INT64PLUS | INT64MINUS | INT64MULT | INT64DIV
          | DATEADD | DATEREM | DATESUB
          | TIMEADD | TIMEREM | TIMESUB
          | TIMESTPADD | TIMESTPREM | TIMESTPSUB
```

Since OCaml does not support function overloading, arithmetic operations (*infix-op*)in QQL are designed to have unique operators (*plus*, *minus*, *mult* and *div*) for each type, such as `+` for integer addition, and `+.` for float value addition, etc. *STRINGPLUS* denotes the concatenation of two strings. For calendar types, like *date*, *time* and *timestamp*, which can be found in the standard OCaml module *Calendar*, QQL provides another three kinds of operations: *sub* calculates the period between two calendar data; *add* increases the calendar data by a specified period and *rem* decreases the calendar data by a specified period.

```
groupby-clause ::= GROUP BY path AS id [COMMA path AS id ]*

having-clause  ::= HAVING h-expr

h-expr ::=
          NOT h-expr
        | h-expr AND h-expr
        | h-expr OR h-expr
        | h-expr IS [NOT] NULL
        | h-expr IN variable
        | h-expr IN LSQRBRACKET h-expr [SEMICOLON h-expr]* RSQRBRACKET
        | h-expr BETWEEN h-expr AND h-expr
        | h-expr LIKE h-expr
        | h-expr infix-cmp h-expr
        | h-expr infix-op h-expr
        | aggr-expr
        | path
        | variable
        | constant
```

The ***group-by*** clause defines criteria for the partition of query result sets into groups, used for the calculation of aggregate expressions. Note, that *group-by* in QQL requires provide an alias for each involved path expression, which can be referred to in clause like *select*, *order-by*, etc. The ***having*** clause specifies a boolean predicate for filtering these groups. The ***having*** allows the use of aggregate expressions, unlike the predicates used in *where* clauses.

```
orderby-clause ::=
                ORDER BY path [direction] [COMMA path [direction]]*

direction      ::= ASC | DESC
```

The ***order-by*** clause identifies the ordering criteria, based on which fields and directions (ascending or descending) the query results should be sorted in. The order of query return data is never guaranteed unless an *order-by* clause is specified. Path expressions specified in the ***order-by*** clause can be constructed from query sources defined in the *from* clause, or aliases specified in the *group-by* and *select* clause.

limit-clause ::= *variable* | *int*

offset-clause ::= *variable* | *int*

Limit and *offset* clauses requires query to return a subset of the result rows. *Limit* specifies the maximum number of rows to be returned; *offset* indicates the number of the start row. Both of these two clauses use an integer value, either constant or variable.

Bulk Data Updating

Qanat provides statements bulk updating of data via QQL; *update* and *delete* queries.

qanat-query ::=
[*select-clause*] *from-clause* [*where-clause*] [*groupby-clause*]
[*having-clause*] [*orderby-clause*] [*limit-clause*] [*offset-clause*]
| *UPDATE className SET set-expr* [*COMMA set-expr*]* [*where-clause*]
| *DELETE FROM className* [*where-clause*]

className ::= *id*

set-expr ::= *path EQ expr*

QQL bulk data updating queries can act on only one orm class (*className*). Thus no explicit join expressions are allowed in these queries. The optional *where-clause* specifies on which rows the operations should be performed. The result returned from the execution of these queries is an integer indicating the number of rows affected.

It is worth noting that a bulk data updating query manipulates the relational databases directly, without touching entities managed by Qanat session. This may cause inconsistency between the underlying database and the runtime session caches in the case when some data affected by the update query had been previously loaded into the session cache, and then modified or removed directly by the query. Here the values of the objects in the session cache will no longer correspond to the values in the database.

To avoid this potential inconsistency problem, Qanat takes the strategy of flushing the session cache before the execution of queries, and evicting the session cache after the execution of bulk updating queries. This means that in-memory entities will have been detached from the session, if they are to be used further, must be re-attached (if they have not changed) or reloaded if they have.

6.3.4 Lexer tokens

META ::= "meta"
TYPE ::= "type"
KEY ::= "key"
BIND ::= "bind"
ORM ::= "orm"
VAL ::= "val"
AND ::= "and"
MODULE ::= "module"
UNIQUE ::= "unique"

LOAD ::= "load"
SAVE ::= "save"
DELETE ::= "delete"
SAVE_UPDATE ::= "save_update"

STRUCT ::= "struct"

```
END        ::= "end"
MUTABLE    ::= "mutable"
WITH       ::= "with"
ON         ::= "on"
MANAGE     ::= "manage"
AUTO       ::= "auto"
ASSIGN     ::= "assign"
REFER_TO   ::= "refer_to"

ALL             ::= "all"
DELETE_ORPHAN   ::= "delete_orphan"

NEW        ::= "new"
WHEN       ::= "when"
OF         ::= "of"
VAL        ::= "val"
OPTION     ::= "option"

INT        ::= "int"
INT32      ::= "int32"
INT64      ::= "int64"
FLOAT      ::= "float"
NUMERIC    ::= "numeric"
BOOL       ::= "bool"
CHAR       ::= "char"
STRING     ::= "string"

DATE          ::= "date"
TIME          ::= "time"
TIMETZ        ::= "timetz"
TIMESTAMP     ::= "timestamp"
TIMESTAMPTZ   ::= "timestamptz"

RESIZESET     ::= "resizeSet"
RESIZEARRAY   ::= "resizeArray"
IDX           ::= "idx"


VALOF      ::= "VALOF"     | "valof"
ITEMSOF    ::= "ITEMSOF"   | "itemsof"

SELECT     ::= "SELECT"    | "select"
AS         ::= "AS"        | "as"
DISTINCT   ::= "DISTINCT"  | "distinct"
COUNT      ::= "COUNT"     | "count"
MAX        ::= "MAX"       | "max"
MIN        ::= "MIN"       | "min"
SUM        ::= "SUM"       | "sum"
AVG        ::= "AVG"       | "avg"

FROM       ::= "FROM"      | "from"
JOIN       ::= "JOIN"      | "join"
LEFT       ::= "LEFT"      | "left"
RIGHT      ::= "RIGHT"     | "right"
INNER      ::= "INNER"     | "inner"
```

```
WHERE      ::= "WHERE"     | "where"
AND        ::= "AND"       | "and"
OR         ::= "OR"        | "or"
LIKE       ::= "LIKE"      | "like"
IN         ::= "IN"        | "in"
BETWEEN    ::= "BETWEEN"   | "between"
IS         ::= "IS"        | "is"
NOT        ::= "NOT"       | "not"
NULL       ::= "NULL"      | "null"

GROUP      ::= "GROUP"     | "group"
HAVING     ::= "HAVING"    | "having"
ORDER      ::= "ORDER"     | "order"
BY         ::= "BY"        | "by"
ASC        ::= "ASC"       | "asc"
DESC       ::= "DESC"      | "desc"

UPDATE     ::= "UPDATE"    | "update"
DELETE     ::= "DELETE"    | "delete"
SET        ::= "SET"       | "set"

uid  ::= ['A'–'Z']+['A'–'Z' | 'a'–'z' | 0–9 | '_']*
id   ::= ['a'–'z']+['a'–'z' | 0–9 | '_']*

QMARK         ::= '?'
DOT           ::= '.'
SHARP         ::= '#'
COMMA         ::= ','
COLON         ::= ':'
SEMICOLON     ::= ';'
VERTICALBAR   ::= '|'
UNDERSCORE    ::= '_'
AMPERSAND     ::= '&'

STRINGPLUS  ::= '∧'
STAR        ::= '*'
PLUS        ::= '+'
MINUS       ::= '-'
DIV         ::= '/'
INT32PLUS   ::= '+:'
INT32MINUS  ::= '-:'
INT32MULT   ::= '*:'
INT32DIV    ::= '/:'
INT64PLUS   ::= '+::'
INT64MINUS  ::= '-::'
INT64MULT   ::= '*::'
INT64DIV    ::= '/::'
FLOATPLUS   ::= '+.'
FLOATMINUS  ::= '-.'
FLOATMULT   ::= '*.'
FLOATDIV    ::= '/.'
NUMPLUS     ::= '+/'
NUMMINUS    ::= '-/'
NUMMULT     ::= '*/'
NUMDIV      ::= '//'
DATEADD     ::= '+~'
```

```
DATEREM     ::= '-~'
DATESUB     ::= '--'
TIMEADD     ::= '+^'
TIMEREM     ::= '-^'
TIMESUB     ::= '--^'
TIMESTPADD  ::= '+!'
TIMESTPREM  ::= '-!'
TIMESTPSUB  ::= '--!'

EQ   ::= '='
NEQ  ::= '!=' | '<>'
GT   ::= '>'
GE   ::= '>='
LT   ::= '<'
LE   ::= '<='

LDQUOTE      ::= “
RDQUOTE      ::= ”
LBRACKET     ::= '( '
RBRACKET     ::= ')'
LSQRBRACKET  ::= '['
RSQRBRACKET  ::= ']'
LBRACE       ::= '{'
RBRACE       ::= '}'
LANGLE       ::= '<'
RANGLE       ::= '>'
LEFTARR      ::= '<='
RIGHTARR     ::= '=>'
```

6.4 Summary

Based on the approaches proposed in the previous chapters, we implemented an object-relational mapping framework in the OCaml programming language, called Qanat. In addition to basic ORM facilities, such as proxy-based lazy loading, cascading operations, automatic modification flushing, and navigational and set-oriented querying, Qanat features compile-time type safety. That is, the embedded ORM annotations and query expressions are type checked against the relational database schema during the preprocessing or compile time. Within the OCaml context, The Qanat query language (QQL) supports the type-safe incorporation of variables, composition of queries, and return of explicitly typed results. The types of these objects are inferred by the compiler in a Standard ML way and type mismatch errors between the database schema, the query language and the surrounding OCaml code is caught at compile time. Qanat also includes a higher-order transaction interface, which avoids the verbose and error-prone try-catch-finally mechanism as well as the external XML type problems and errors due to breaking code naming conventions that occur in popular Aspect Oriented Programming approaches to transaction support.

In this chapter, we presented the formal grammar of Qanat language extensions for the embedding of ORM metadata and QQL in section 6.3, and give a tutorial demonstrating how to program in Qanat in section 6.2. The helloworld example introduced the basics of Qanat programming; while the bookshop example demonstrates the full feature set of Qanat, showing the mapping of various relationships in orm classes, including *many-to-one*, *one-to-many*, *many-to-many* and *one-to-one* relations, and the use of QQL language.

Chapter 7

Evaluation and future work

In this chapter, we present our analysis of the Qanat framework, and show the result of benchmark comparisons between Qanat applications and PGDriver (a JDBC-like, low level OCaml binding for PostgreSQL databases, implemented as part of and distributed with Qanat) applications. In the last part of this chapter, we discuss future work.

7.1 Qanat analysis

For the purpose of our analysis of our approach and of the Qanat framework itself, we borrow criteria identified by Cook and Ibrahim [CI05]. As described in Chapter 1, the analysis criteria covers various aspects of an object-relational implementation from typing, to static type checking, interface style, code reuse, concurrency and transactions, and optimisation.

Typing: The key problem of the object-relational impedance mismatch is the difficulty in aligning the types between programming languages and relational databases. Both of them support primitive types and data structures, but programming languages model data as objects or other composite structures, such as lists, records and tuples, and associate them by using references, while relational databases store data inside tables and represents relationships via foreign key constraints.

To ameliorate the mismatch, Qanat acts as an intermediate mapping layer for data conversion between the OCaml language and the relational database. The mapping of primitive-type values is predefined in Qanat, as presented in Figure 3.6. Qanat supports mapping relational tables to object-oriented classes, table foreign keys to object references. Furthermore, it supports the mapping between OCaml composite structures (e.g. record, tuple, variant) and a group of table columns. These object-relational mappings can be generated from the database by using tools `ORMGen` and `DBSchema`, or can be written, or just customised, by programmers. Based on these mappings, operations invoked on OCaml values are reflected to the underneath database without extra effort. Qanat uses the *option* type to handle NULL in SQL.

Static typing: This refers to the ability to detect type errors at compile time instead of deferring errors until run time.

Program compilers in statically typed languages guarantee the code, at compile time, to be executable without type errors, but cannot detect mismatches between programs and external databases, especially when queries are expressed as literal strings. In Qanat, we embed ORM mapping metadata and QQL directly in the program, and check the semantics of the metadata against additionally loaded database schema at the compile time (section 3.2.2). For the type safety of embedded queries, we introduced type avatars, a dummy data structure, accompanying each query but only used for the capture of query type constraints. SQL overloads some functions, e.g. sum works on integers and on floats and returns a value which is the same type as that of the values it has operated on. To infer the types of such results properly, we need to capture these type constraints more precisely than the normal type system for SQL allows. We use phantom types to

do so. By employing type avatars and phantom types, Qanat extends compile-time type checking to embedded queries (Chapter 4). Furthermore, OCaml type inference with avatar functions and phantom types enables Qanat to return explicitly typed query results and inference the type of query variables at compile time. Programming in Qanat means that fewer modify-build-test cycles would be required and type errors are caught at the compile time.

Interface style: Qanat supports saving and querying entities in the following styles,

- **Navigational:** Qanat supports loading and persisting values simply by navigating the in-memory query graph of entities (persistent objects) and updating those objects exactly like any other OCaml object. Once the user has obtained an entity from the database (either by an explicit load request, a query or by creating an object and attaching it to the database, modifications to the object are transparently persisted to the database, objects that this object is made to refer to are persisted as well, and following object references will transparently, and lazily, load the corresponding objects from the database. This lazy loading is achieved by using proxies (section 3.3). The session instance tracks modifications of managed entities, and flushes these updates into the database when the session is about to close. Thus Qanat provides orthogonal ORM persistence.
- **Set-Oriented:** Qanat also supports explicit querying by providing an embedded query language (QQL). QQL allows programmers to interact with the database in a way similar to SQL but using object-oriented classes, objects and fields to write queries. The execution of QQL queries returns explicitly typed values or entities, and puts these returned entities under the management of the session instance, so that they can be further used, either in other QQL queries in a set-oriented style or in Qanat's navigational style.

Reuse: QQL supports defining parameterised queries for multiple execution by passing different parameter values on each invocation, and complicated queries by composing existent queries. In the latter case, existent queries are passed as variables (i.e. subqueries) defined in another query to construct a new query; the type of these variable are inferred in a standard OCaml way based on the query context, thus any type mismatch between the variable definition and the passed query or value causes compile-time type mismatch error. Using parameterised and composite queries reduces unnecessary repetition in programs.

Concurrency and transactions: Data-driven applications typically use an exception mechanism or aspect-oriented programming to provide transaction management. In Qanat, we leverage OCaml's higher-order functions to provide an alternative and simple solution for both single and multiple transactions. The single transaction handler invokes a use case function on a single database source. The multiple transaction handler provides the ability of invoking functions that span over several database sources (see Chapter 5). Both of them start by requesting a valid transaction (a new transaction will be started if no valid transaction exists) from the session factory, then performing the use case function. If the use case initiated the transaction (rather than merely having been invoked from within another use case function in the transaction) then the transaction will be automatically committed if the use case returns successfully. Otherwise, if an exception was raised, it will be rolled back, again automatically. Again compile-time typing and session construction approaches used ensure that many of the common transaction programming errors, that are only found in JDBC or AOP based approaches at runtime, are either caught at compile time in Qanat or are not possible to begin with.

Optimisation: A proper optimisation strategy may significantly reduce the running time of query execution in Qanat. Since Qanat was designed and developed as a demonstrator for our research on compile-time type safe persistence, we have not put significant effort into maximising its performance. However, we have also avoided imposing design choices that would require algorithm that would exclude performance comparable, at least, to Hibernate. Although we present the translation of QQL into equivalent SQL in section 4.9, the optimisation and execution performance of the generated queries remain open problems for future work.

7.2 Benchmark comparison of Qanat and PGDriver

The principal of the benchmark is to measure the differences of Qanat applications and raw database applications, especially the overhead cost added and over raw database operations when using Qanat's object-relational mapping to implement data persistence. We run a number of tests comparing the equivalent code in Qanat and PGDriver. The purpose of these tests are to determine if the overhead imposed by Qanat on database applications is unacceptable for real world applications.

PGDriver is a JDBC-like, raw database library for PostgreSQL in OCaml. It was developed as a sub project for Qanat. The PostgreSQL library was required as currently available PostgreSQL libraries for OCaml provide only limited support for transactions and for query parameter escaping and they force loading entire query results eagerly from the database, even if these results are very large, rather than meeting our requirements of loading results in blocks, lazily on demand.

The benchmark consists of testings for data insertions, loadings and updates, based on a one-to-many relationship data model. The model contains a number of published news items and each news item has a number of comments. The insertion, loading and update occurring on the news will be cascaded to its comments. The benchmark system logs the time of code execution. The source code can be found in the Qanat package.

We also compare the lines of code and error types reported at compile time between Qanat and PGDriver applications. These results directly affect the development time and runtime robustness of a data-driven application.

7.2.1 Lines of codes

The use of Qanat as the data persistence implementation results in shorter programs and implies shorter development time. Figure 7.1 depicts the comparison of lines of code in the equivalent Qanat and PGDriver projects. Because Qanat provides a transparent mapping between programs and underlying databases, it avoids requiring users to write complex and verbose SQL statements and query management code to handle the data persistence, retrieval and relationship management, letting users focus instead on the use cases. The lines of code (LOC) for Qanat projects is shorter than those written in PGDriver by about about 40% – 45%, and this saving tends to increase with the complexity of applications: the more complex database relations and use cases, the greater the percentage savings that the Qanat project offers over the raw database project in terms of lines of code.

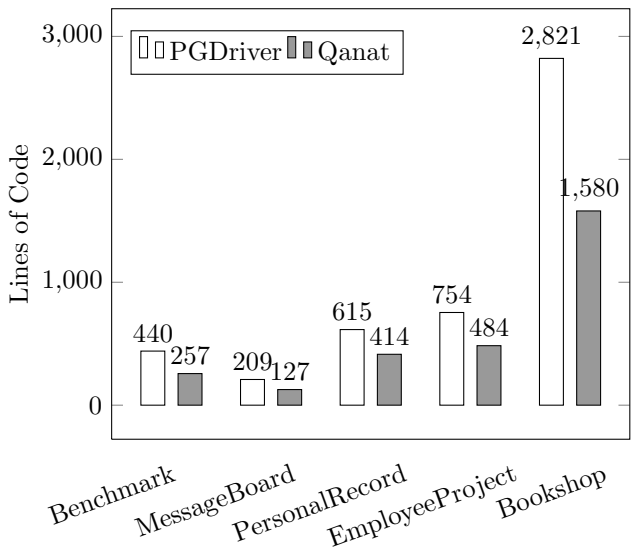


Figure 7.1: Lines of Code in Qanat and PGDriver applications

The LOC comparison is performed over a number of database projects,

Benchmark: based on a one-to-many relationship model, news and its comments. It is developed for the testing of Qanat insertion, loading and update performance.

MessageBoard: saves messages and their reply messages. The basic functionality includes posting new message, replying, updating and deleting old messages.

PersonalRecord: a system recording personal details. Each person is recorded with a detail record consisting of various information in various types. The relationship between a person and his/her detail record is defined as a weak one-to-one relationships; i.e. the removal of a person triggers the removal of the corresponding detail record, both of which share the same primary key.

EmployeeProject: used to record employees and their related projects, the relation is many-to-many: employees can join more than one project and each project can have several members.

Bookshop: an interactive platform for daily bookshop use. This project demonstrates all kinds of functionality of Qanat, including the use of QQL. This project is discussed in Chapter 6.

7.2.2 Performance testing

The performance testing is based on the news and comments benchmark, and the main purpose is to measure the time cost taken when using Qanat to manage the relationship. For each test, we run a number of test cases by increasing the number of associated comments for each news. This directly increases the work of relation management in Qanat.

The graphs below show the result of the performance comparison. We have to admit that, because of lacking optimisation strategies in Qanat implementation, the efficiency of Qanat is quite low compared to raw database operations. But it is as expected that the increase of time cost remains in a smooth curve when the number of association relations is relatively small, and becomes sharp at the point when the cost of relation management and modification tracking is greater than the cost of basic database interactions.

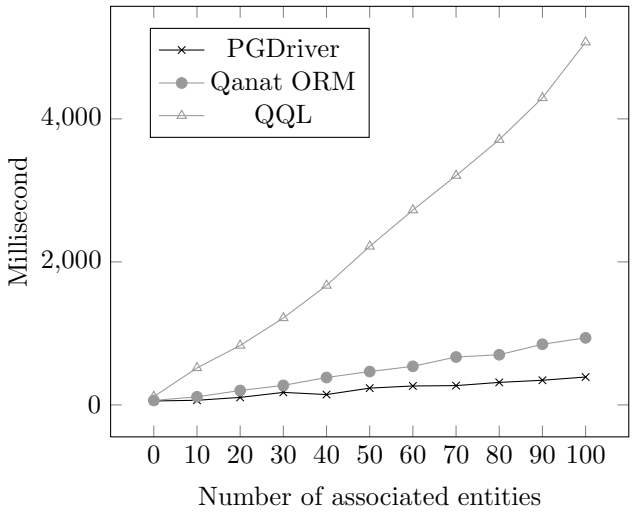
Figure 7.2a depicts the result of retrieving a number of news items and their associated news items in three different ways, i.e. using the direct database operation provided by PGDriver, the ORM load facility in Qanat, and QQL queries, respectively. The time costs of data retrieval in PGDriver and Qanat ORM are quite close. This result is based on the fact that we do not count the time for session flushing, since there is no modifications to the database occurring during this process. The additional time cost in ORM operations is mainly caused by the process of construction of objects and associated proxies from the result set and checking for data existence in the session's caches. This additional cost increases with the number of loaded entities. It is worth noting that this particular loading test doesn't load repetitive data from the database, thus the cache checking in the session always returns false. Otherwise, some database hits might be avoided.

The time cost of QQL query loading increases with the number of associated entities, but the efficiency is below our expectations. Our analysis suggests that the main reason for this result is:

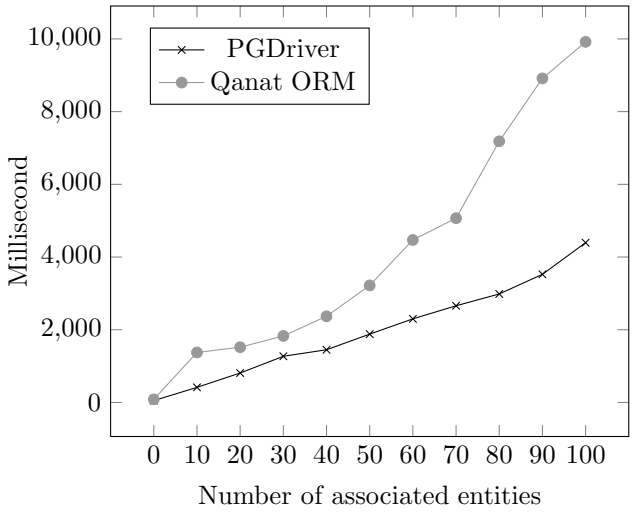
1. Our implementation of QQL does not incorporate any performance optimisation, nor consider the execution performance. In our current implementation, the invocation of a query generates SQL from QQL and executes it by using standard statement rather than the prepared statement. A significant efficiency problem occurs when executing one query multiple times by passing different parameter values, such as the test in our case. This is is an engineering issue that could be easily corrected in the future.

2. the execution of each QQL query invokes session flushing, which performs dirty data checking, manages the relationships and writes modifications back to the database. This flushing action is required to ensure return the latest version of data from the database. We can envisage many cases when this flushing is either not necessary or could be limited to only sections of the session cache. Such optimisations are very feasible but have been left to future work.

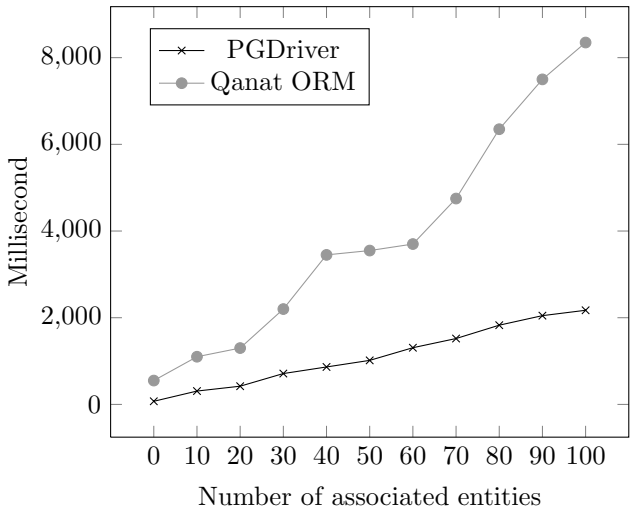
3. a relatively small cost is in adding loaded entities into the session, this involves the additional action of constructing data copies for managed entities.



(a) Loading test



(b) Inserting test



(c) Updating test

Figure 7.2: Comparative cost of data insertion in Qanat and PGDriver

The results of insertion and updating tests are depicted in Figure 7.2. We note that the time cost of Qanat operations increase significantly when the number of associated entities reach a point at which the cost of relation management and modification tracking exceeds the cost of basic database interactions. Compared to saving a new (transient) entity into the database, updating an existing persistent entity to the database is much faster. This is because, if the entity has a `auto` generated primary key, this key has to be generated by executing an extra database query. However, there is a simple optimisation that can significantly improve this case: when a new `auto` key is needed, obtain a number of them in one query, use one for the current entity and cache the rest for future use. Thus the average cost of auto key generation is reduced to a fraction of its original cost.

7.2.3 Compile-time detected errors in Qanat

The performance overhead of using the current version of Qanat over using PGDriver is not negligible. We believe that significant work on optimisation of Qanat's strategies could greatly reduce this overhead. However, except in special cases, it is unlikely that this overhead would ever be insignificant. The question that remains, therefore, is whether the benefits of Qanat in other areas might outweigh the disadvantages with respect to performance.

Qanat applications benefit from an increase of development productivity and compile-time type safety. We have already shown that the LOC of Qanat projects is between 40% and 45% less that that raw PGDriver applications. By employing techniques presented in the previous chapters, Qanat enables detection of type errors existing in both ORM and QQL code at compile time. This means fewer modify-build-test cycles would be required during the development, less expensive testing required at all, and fewer run time errors possible in the deployed system, hence, we argue, a significantly cheaper and more robust product.

No.	Error Type	Description
00	Syntax Errors	not listed here
Qanat ORM Related		
01	Nonexistent table	using nonexistent table in orm classes
02	Nonexistent column	using nonexistent column in orm classes
03	Nonexistent foreign key	referring to nonexistent foreign keys in orm classes
04	Mismatched primary key	the primary-key field mismatches table primary-key
05	Mismatch field type	field type mismatches the column(s) type
06	Duplicated class field	assigning the same name to more than one field
07	Undefined key or bind	using undefined metadata key/bind in the mapping
08	Undefined orm class	referring to an undefined orm class
09	Undefined record/variant	using record/variant type that lacks mapping metadata
10	Null exception	errors related to the mapping of nullable columns
QQL Related		
11	Undefined orm class	using undefined orm class in the query
12	Mismatch types	using variables/fields in a wrong context
13	Duplicated alias names	assigning an alias name to multiple classes/variables
14	Unbound values	using undefined values in the query context
15	Invalid query source	querying on class that is not an orm class
16	Invalid join operation	joining two non-queryable fields
17	Invalid query return	returning elements not referred to in group-by
18	Invalid group-by element	defining group-by on collection type fields
19	Invalid order-by element	using invalid elements in order-by

Figure 7.3: Compile-time detected errors in Qanat

Figure 7.3 lists the main type errors that can be detected by Qanat at the compile time. In

addition to ORM and QQL syntax errors, these errors can be categorised as

object-relational mapping metadata errors:: These are reported in the case when the mapping metadata does not match the database schema, such as, the mapped tables, columns, or foreign keys cannot be found in the schema.

orm class definition errors:: These occur when the program tries to use undefined metadata key/bind or unmapped data structures, such as classes/records/variants that are not declared as ORM types, in the orm class definition.

type errors in QQL: because of using type avatars and phantom types, the type checking of QQL queries is performed by the program compiler, and errors are reported as type errors, such as using unbound values in the query, or applying aggregate functions to non-collection values.

7.3 Limitations and future work

The Qanat ORM framework demonstrates that compile-time type safety is achievable in the OCaml language. However, as shown in section 7.2.2, the performance of ORM operations is a problem for the practical use of Qanat in large applications that involve frequent and large amounts of data retrieval and updating.

We have identified the following limitations and issues for future work in our approach:

- Dynamic changes to the database schema from foreign programs

  Qanat has the capability of detecting mismatches between the orm class and the database schema at compile time. It also supports the mechanism of checking the consistency of the database schema and the program at runtime when the first session instance is initialised from the session factory. However, this does not preclude the possibility of type mismatches occurring at runtime. A separate program could access the database and modify the database schema while the Qanat program is running. In such a case the Qanat program, when it discovers such a runtime type mismatch error, will throw a runtime exception.

  We have not discovered satisfactory approaches to deal with such problems. We believe that, for databases that support such features, the use of database triggers that can be fired when a schema change occurs, together with a call-back mechanism that can be used by the database to inform clients of the schema change, would help to at least notify the clients of the potential problem. Nonetheless, if the schema change is incompatible with the schema the Qanat program was compiled with, the Qanat program can do little other than throw a runtime exception.

- Internal dynamic changes to the database schema

  There is no support in Qanat for its own dynamic changes to the database schema while a Qanat program is running. The Qanat approach is predicated on a statically type checked philosophy, and dynamically changing the types during execution is not compatible with this approach.

  Nonetheless, there is a limited form of dynamic schema change that would be compatible, although Qanat does not currently support it. This is where parts of the database schema are fully statically typed but not existent in the actual database at initial runtime. Theoretically, the system could type check all necessary queries at compile time, but, at runtime, would have to explicitly check for the existence of the appropriate schema parts before interacting with them. This would allow scenarios where, for example, a query could generate a temporary table as part of a complex transaction, use this table in further operations, and then drop the table. We believe there is some scope for future work on this idea, but have not, as yet considered it in detail.

- Database procedures and triggers

  Relational databases allowing creating stored procedures and database triggers. They can be explicitly invoked by issuing an SQL statement in the case of stored procedures for running

daily subroutines, or implicitly invoked in the case of database triggers as the response action to various predefined events. These database procedures and triggers are defined using SQL. Qanat does not currently support database procedures or triggers and we leave it to future work to investigate the many issues that adding such support in a compile-time type safe way would entail.

- Check constraints

  A check constraint is a predicate applied to each row in the table for guaranteeing the "business logic" validity of data when adding or updating a record in a table. Qanat currently doesn't have a direct solution to adding check constraints in orm classes.

- Support for new types and functions

  The current Qanat framework has been developed for the use with PostgreSQL. It supports most standard SQL data types, but there are still some uncommon data types that are not supported, such as the binary string type, network address type, etc. Based on these types, a group of functions are defined in the PostgreSQL's version of SQL. Adding support for these data types needs a few modifications on Qanat source code, especially for designing the corresponding shadow phantom functions for these newly introduced types and functions.

- Re-entrancy

  The current version of Qanat is not re-entrant, mostly because of the use of a global proxy weak hash table, so that persistence support code can determine whether a entity is an entity reference is to an initialised proxy or not. There are, however, some other sources of re-entrancy problems. We have not yet investigated the issues of making Qanat re-entrant and leave this to future work.

- Automatic version support for long transactions

  A standard pattern supporting long transactions in ORM frameworks is to implement them as a series of short transactions connected by using a version controlled detached entity that is re-attached in succeeding short transactions. With automatic version support, the re-attachment fails if the database version of the entity has been modified between successive short transactions. In this case a compensating action can be taken to undo the effects of the previous short transactions. This requires support for version control on entities, which is not part of the current Qanat system.

  There are no technical problems with adding such support, which we plan to do for a future release, but, in the mean time, version support can be simulated by the application programmer at the price of some small inconvenience.

- Composite column foreign-key relationships

  Qanat supports various types of relationships, including *one-to-many*, *many-to-one*, *many-to-many* and *one-to-one*, and various types of primary-key fields, either mapped to single column or multiple columns. It currently lacks support for composite column foreign-key relationships i.e. table relationships defined by using a foreign key that consists of more than one column, although it is not difficult to add the support of this kind of relationship to Qanat. Again, we intend to do so for a future release.

- Global caching for performance improvement

  The session structure includes a number of basic caches to ensure that consistent objects are returned for multiple entity request and to support modification tracking when the session is about to close. Such a structure can provide a simple caching mechanism, but the effect is limited, since these caches are bound with the database transaction and closed when the transaction finishes.

  Incorporating a global caching system that provides caching services for multiple transactions, similar to Hibernate's second level cache, may provide significant improvements for some situations. The loading of entities would then perform a check in the cache first, helping avoid unnecessary database hits.

- ORM definition completeness

  Even though Qanat supports program relationships like one-to-many, many-to-one, one-to-one and many-to-many in bidirectional or unidirectional way, it does not yet support mapping a relationship where any of the foreign keys involved consist of multiple columns. The solution of this problem requires extending the structure of metadata *key* and *bind* to contain more column information.

- Adaptors for various database connections

  Qanat includes a raw database library for PostgreSQL, called PGDriver, and a database layer to provide a uniform database interface. By implementing a database-specific adaptor it becomes possible to switch the application from one kind of database to another, since database operations are written in a uniform manner, and the driver can be chosen at runtime from a collection of compiled in drivers. The implementation of the adaptor requires wrapping database operations in a record type structure, the definition of which is predefined in the database layer. As an example, we implemented the adaptor for PostgreSQL. Adaptors for other databases can be implemented in a similar way.

- QQL optimisation

  An open problem we are facing in QQL is the optimisation of QQL translation,

  In section 4.9, we presented the approach of translating QQL into equivalent SQL by leveraging monoid comprehension as the intermediate representation of query constructs. But we did not consider translation optimisation at this time. Thus a query constructed by using query composition results in a SQL string consists of multiple nested sub queries, rather than an efficient, equivalent, plain join SQL query. In most cases, nested sub queries can be transformed into unnested plain SQL. Approaches for object-oriented query language translation and optimisation have been discussed in the literature [KKM94, GS96, Gru99, FM00, Gru03]. Although experience can be obtained from this research, further work is required for the optimisation of our QQL.

- QQL completeness

  Even though QQL supports a large variety of queries, it does not provide all the functionality that full SQL does. For example, QQL does not currently support the SQL intersection operation and one cannot therefore write a query that composes two `unchanged` existing queries with an intersection operation. Of course, one can rewrite the query so as to compute the intersection in the usual nested query way, but that does not use the existing sub queries unchanged.

  Another limitation in the current QQL is that defining a query with sub-queries without using query composition is not supported in the syntax. Using sub-queries, nested to any depth, is supported in QQL, but only by declaring a variable of type query in the from clause, and then passing the sub-query as a parameter. We do not anticipate any significant difficultly in adding non-compositional support for defining nested queries.

7.4 Summary

In this chapter, we evaluated Qanat framework by running a benchmark to measure the performance of Qanat database operations, and analyse the development effort of several Qanat projects. We admit that the current version of Qanat suffers a significant performance overhead, compared to raw database operations provided by PGDriver. However, the improvement in the Qanat project development is also significant. The lines of code of a Qanat project can be cut to nearly 55% of its equivalent raw database application. And the compile-time type safety ensures that fewer modify-build-test cycles are required and many common runtime errors are no longer possible.

Qanat ORM and QQL demonstrate the applicability of the approach we investigated. however, there are still open problems hindering it from becoming a practical, efficient object-relational solution. In the latter part of this chapter, we have identified problems that address this issue for future work.

Chapter 8

Conclusions

In this thesis we investigated the problem providing a compile-time type-safe object-relational mapping framework in the language OCaml. Because of the object-relational impedance mismatch problem, it is difficult to quickly develop reliable, well-engineered database applications. A number of approaches have been proposed to make such integration simpler, such as using object relational mapping (ORM) frameworks like Hibernate, JPA, and Ruby-on-Rails, etc.

Our approach has been driven throughout by the fact that these frameworks have not been particularly concerned with compile-time type safety. Type mismatch errors between the program and the database schema occur quite often during program development, and the techniques used in these frameworks often defer error checking on database operations until runtime.

We argue that OCaml's strict type checking at compile time, type inference, phantom types, higher-order functions, polymorphism, data abstraction, and the preprocessing technique, provide the opportunity to build a better solution to this problem. By leveraging these techniques, we achieved the effect of compile-time type safe, object-relational mapping and domain specific query language, and higher-order transaction handling.

While the overall design of the ORM is very similar to Hibernate, such as using proxies for lazy loading, and a session structure to manage persistent objects, etc, many of the details are different because of the need to satisfy compile-time type safety and because of the different problems, constraints and opportunities that the use of OCaml, rather than Java, implies. Instead of using external XML mapping file, we embedded the mapping metadata in the program by extending the language syntax via a pre-processing technique, and read database schema information during compilation, using which the extended ORM class definition is translated into standard language code with proper type information. Moreover, additional runtime schema checking is employed to minimise the risks of executing programs on an a database whose schema has been modified since the program was compiled.

We embedded, in OCaml, a compile-time type-safe, object-oriented querying language (QQL), which embraces the functionality of SQL, and also introduces object-oriented features that tend to result in shorter and more intuitive queries by using implicit join, query composition and nested results. This query language is embedded in the program, written in a syntax adapted from standard SQL. To ensure the compile-time type safety of QQL, we proposed the approach of using type avatars, a dummy structure, to capture the type constraints of queries, and employing phantom types to achieve type safe function overloading for the support of aggregation functions and other query functions. Furthermore, we also proposed a partial type inference algorithm based on type equations, and compiled queries into a self-contained query structure, to incorporate variables in queries in a type safe manner and to compose queries as components, also type safely. By leveraging monoid comprehension as the intermediate query representation, QQL queries are translated into equivalent SQL strings that are delivered to and executed in the database server.

Unlike mainstream orm frameworks, such as Hibernate and LINQ, the majority of the work of our approach is performed using pre-processing before the compilation. Because of this, we faced a number of unexpected problems, such as having to perform type inference for variables without knowing the concrete types of field expressions, and needing to generate function expressions for unknown query result types. For the first problem, we implemented a Hindley-Milner-like type

constraint inference algorithm based on type equations (section 4.7). Instead of inferring the concrete type for each query variable, this algorithm finds an equivalent type for each query variable, thus those functions applicable to the equivalent type values can be applied on this query variable. For the latter problem we proposed a practical technique (section 4.8), that is, "phantom" functions for classes and encapsulated fields of orm classes and queries are generated in a standard module, and wrapped and organised inside an object graph which mirrors the structure of the true orm and query classes and fields. This allows to, even in the presence of separate compilation, generate code at pre-compiler time that will compile correctly if and only if the orm and query types expected by the QQL queries actually match the true orm and query types. Further, it allowed us to support a powerful navigational query expression mechanism in the queries and a sophisticated dynamic query composition mechanism without sacrificing compile-time type safety.

Rather than using the verbose and error prone exception patterns of JDBC or the less verbose but still error prone aspect-oriented programming approach of Spring, OCaml enables use to use higher-order functions and data abstraction to provide a type-safe and concise transaction interface. We proposed a transaction interface for handling transactions over a single or multiple database sources. The higher-order feature allows programmers to write use cases in a separate function, then easily use our proposed transaction interface to invoke the use case within transaction scope, which automatically commits when the use case execution finishes, or rolls back when any exception occurs. With such an approach, we achieved the separation of the use case function from transaction logic.

We implemented an ORM framework named Qanat based on our proposed approach. It includes functionality such as compile-time ORM, QQL and a higher-order transaction interface. Qanat provides a set of tools, such as `DBSchema`, which extracts the database schema from the specified database into a JSon format file, and `ORMGen`, which generates orm definitions from a database schema file. The transparent ORM solution reduces significantly the amount of code required for achieving equivalent functionality over a raw JDBC-like approach. The compile-time type safety ensures that fewer modify-build-test cycles and less testing in general are required and a more robust program is deployed.

Qanat ORM and QQL demonstrate the applicability of the approach we investigated, however, there are still open problems hindering it from becoming a practical, efficient object-relational solution. QQL faces a serious efficiency problem because of a lack of work on optimisation. Handling of stored procedures, user defined procedures stored in the database that are executed on the database server, is also an open problem.

The current, full version of the Qanat framework has been released to open source and is available at `http://www.cs.bham.ac.uk/~aps/research/projects/qanat`.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. PS-algol: A language for persistent programming. In *In Proc. 10th Australian National Computer Conference*, pages 70–79, Melbourne, Australia, 1983.

[ACC82] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. PS-algol: an algol with a persistent heap. *SIGPLAN Not.*, 17(7):24–31, 1982.

[ACC83] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. Algorithms for a persistent heap. *Software Practice and Experience, Vol 13, No 3*, pages 259–272, 1983.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[AJDS96] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system. In *in Proceedings of the Seventh International Workshop on Persistent Object Systems*, May 1996.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, 1995.

[Amb96] Scott Ambler. Object-relational mapping. *Software Development Magazine*, 4(10):47–50, Oct 1996.

[Amb03a] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, USA, 2003.

[Amb03b] Scott W. Ambler. *Overcoming The Object-Relational Impedance Mismatch*. John Wiley & Sons Inc, Hoboken, NJ, 2003.

[Ame99] American National Standard for Information Technology. Database languages – SQLJ – Part 1: SQL routines using the Java programming language. ANSI/INCITS 331.1-1999, 1999.

[AMSS05] Syed Mujeeb Ahmed, Jack Melnick, Neelam Singh, and Tim Smith. *Pro*C/C++ Programmer's Guide, 10g Release 2 (10.2)*. Oracle Corporation, June 2005.

[ASL89] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: a query language for manipulating object-oriented databases. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 433–442. Morgan Kaufmann Publishers Inc., 1989.

[Bac73] Charles W. Bachman. The programmer as navigator. *Communications of the ACM*, 16(11):635–658, 1973.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR*, volume 4790, pages 151–165, 2007.

[BHA+04] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin. Student paper: HaskellDB improved. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM Press, 2004.

[BK04] Christian Bauer and Gavin King. *Hibernate in Action, Practical Object/Relational Mapping*. Manning Publications, 2004.

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1):87–96, 1994.

[BMS05] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in $C\omega$. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, pages 287–311, 2005.

[BMT07] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C#. *SIGPLAN Not.*, 42(10):479–498, 2007.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200. ACM Press, 1998.

[BSSS06] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 20–31, New York, NY, USA, 2006. ACM.

[BT00] Gavin Bierman and Niki Trigoni. Towards a formal type system for ODMG OQL. Technical Report TR 497, University of Cambridge, Computer Laboratory, October 2000.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM.

[CC06] Sylvain Conchon and Evelyne Contejean. The Alt-Ergo automatic theorem prover, 2006.

[CD08] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 13–22, New York, NY, USA, 2008. ACM.

[CFS08] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In Marco T. Morazán, editor, *Trends in Functional Programming*, volume 8. Intellect, 2008.

[CH03] James Cheney and Ralf Hinze. First-class phantom types. Computer and Information Science Technical Report TR2003-1901, Cornell University, 2003.

[CI05] William R. Cook and Ali H. Ibrahim. Integrating programming languages and databases: What is the problem? In *In ODBMS.ORG, Expert Article*, 2005.

[CKF+01] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring operating system aspects: using AOP to improve OS structure modularity. *Communications of the ACM*, 44(10):79–82, 2001.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM, 2001.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

[CM84] George Copeland and David Maier. Making smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, 1984.

[CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml*. O'Reilly France, 2000.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[Coc01] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2001.

[Cod83] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983.

[Con00] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 Specifications: From the W3c Recommendations*. iUniverse, Incorporated, 2000.

[CR05] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106, New York, NY, USA, 2005. ACM Press.

[CR06] William R. Cook and Carl Rosenberger. Native queries for persistent objects, a design white paper, February 2006. `http://www.cs.utexas.edu/~wcook/papers/NativeQueries/NativeQueries8-23-05.pdf`.

[Cro06] Douglas Crockford. RFC 4627 – The application/json media type for JavaScript Object Notation (JSON), July 2006. `http://tools.ietf.org/html/rfc4627`.

[Dah99] Markus Dahm. Byte code engineering. In *In Java-Informations-Tage*, pages 267–277. Springer-Verlag, 1999.

[DDB91] Klaus R. Dittrich, Umeshwar Dayal, and Alejandro P. Buchmann, editors. *On Object-Oriented Database Systems*. Topics in Information Systems. Springer, 1991.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[dR03] Daniel de Rauglaudre. *Camlp4 Reference Manual 3.07*. INRIA, September 2003.

[dt08] Frama-C development team. *Frama-C: Framework for modular analysis of C*, 2008.

[EAK+01] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of AOP. *Commun. ACM*, 44(10):33–38, 2001.

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM.

[FEB03] Maydene Fisher, Jon Ellis, and Jonathan C. Bruce. *JDBC API Tutorial and Reference*. Pearson Education, 2003.

[Feg94] Leonidas Fegaras. A uniform calculus for collection types. Technical Report 94-030, Oregon Graduate Institute of Sciene & Technology, 1994.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[FL04] J.-C. Filliâtre and P. Letouzey. Functors for proofs and programs. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 47–58, New York, NY, USA, 1995. ACM.

[FM00] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[FP06] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. *J. Funct. Program.*, 16(6):751–791, 2006.

[Got74] Eiichi Goto. Monocopy and associative algorithms in an extended Lisp. Technical Report 74-03, University of Tokyo, 1974.

[Gro09] The PostgreSQL Global Development Group. *PostgreSQL 8.4 Official Documentation - Volume V. Internals and Appendixes*. Fultus Corporation, Sep 2009.

[Gru99] Torsten Grust. *Comprehending Queries*. PhD thesis, University of Konstanz, September 1999.

[Gru03] Torsten Grust. Monad comprehensions: A versatile representation for queries. In *The Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, pages 288–311. Springer Verlag, September 2003.

[GS96] T. Grust and M. H. Scholl. Translating OQL into monoid comprehensions - stuck with nested loops? Technical Report 1430-3558, Universitat Konstanz, 1996.

[GSD04] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, Washington, DC, USA, 2004. IEEE Computer Society.

[Gul03] Ceki Gulcu. *The Complete Log4j Manual*. QOS.ch, 2003.

[Haiml] Bruno Haible. *Weak References, Data Types and Implementation*, `http://www.haible.de/bruno/papers/cs/weak/WeakDatastructures-writeup.html`.

[Hal01] Thomas Hallgren. Fun with functional dependencies. In *Proceedings of the Joint CS/CE Winter Meeting*, pages 135–145, January 2001.

[HHJW94] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 241–256, London, UK, 1994. Springer-Verlag.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, July 2002. Technical Report.

[Hib07] Hibernate. *Hibernate Annotations Reference Guide 3.3.0*, 2007.

[HK03] Yin-Fu Huang and Wen-Feng Kuo. The query translation between object-oriented and relational databases. *Journal of the Chinese Institute of Engineers*, 26(5):715–720, 2003.

[HU79] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):383–385, 1989.

[Ibr92] Mamdouh H. Ibrahim. Reflection and metalevel architectures in object-oriented programming. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 315–318, New York, NY, USA, 1992. ACM.

[ISO] ISO/IEC 9075-3:1995: Information technology – database languages – SQL – Part 3: Call-level interface (SQL/CLI).

[JA99] Mick J. Jordan and Malcolm P. Atkinson. Orthogonal persistence for Java? - a mid-term report. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3)*, pages 335–352, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Jam05] Martin Jambon. How to customize the syntax of OCaml, using Camlp4, 2005. `http://martin.jambon.free.fr/extend-ocaml-syntax.html`.

[JHA+05] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Colin Sampaleanu. *Professional Java Development with the Spring Framework.* Wrox Press Ltd., Birmingham, UK, 2005.

[JHD+09] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, A. Arendsen, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, M. Fisher, S. Brannen, R. Laddad, A. Poutsma, C. Beams, T. Abedrabbo, and A. Clement. *The Spring Framework - Reference Documentation (3.0.0)*, 2009. `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html`.

[Jin99] Beihong Jin. Translating object query language. In *TOOLS '99: Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, page 380, Washington, DC, USA, 1999. IEEE Computer Society.

[Joh02] Rod Johnson. *Expert One-on-One J2EE Design and Development.* Wrox Press Ltd., 2002.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[Jon07] Richard W.M. Jones. PG'OCaml, February 2007. `http://merjis.com/developers/pgocaml`.

[JW07] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 61–72, New York, NY, USA, 2007. ACM.

[KKM93] Daniel A. Keim, Hans-Peter Kriegel, and Andreas Miethsam. Object-oriented querying of existing relational databases. In *In Proc. Intl. Conference on Database and Expert Systems Applications (DEXA*, pages 325–336, 1993.

[KKM94] Daniel A. Keim, Hans-Peter Kriegel, and Andreas Miethsam. Query translation supporting the migration of legacy databases into cooperative information systems. In *Proceedings of the Second International Conference on Cooperative Information Systems*, pages 203–214, 1994.

[KL89] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Press and Addison-Wesley, 1989.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, June 1997.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[KR90] M.W. Kisworo and P. Rajagopalan. Implementation of an object-oriented front-end to a relationaldatabase system. In *TENCON 90. 1990 IEEE Region 10 Conference on Computer and Communication Systems*, volume 2, pages 811–815, Sep 1990.

[KS06] Mike Keith and Merrick Schincariol. *Pro EJB 3: Java Persistence API (Pro)*. Apress, Berkely, CA, USA, 2006.

[KW92] David J. King and Philip Wadler. Combining monads. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 134–143, London, UK, 1992. Springer-Verlag.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, New York, NY, USA, 1998. ACM.

[LDG+07] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.10, Documentation and user's manual*. INRIA, Oct 2007.

[Lei03] Daan Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, November 2003.

[Ler97] Xavier Leroy. *The Objective Caml system, release 1.07, Documentation and user's manual*, 1997.

[LM99a] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proc. 2nd Conference on Domain-Specific Languages'99*, pages 109–122. ACM Press, 1999.

[LM99b] Daan Leijen and Erik Meijer. Translating do-notation to SQL, 1999.

[LS76] B.W. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, 1976.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *LNCS 3780*, pages 139–160, Nov 2005.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM.

[MBC+88] R. Morrison, A. L. Brown, R. Carrick, R. Connor, and A. Dearle. On the integration of object-oriented and process-oriented computation in persistent environments. In *Lecture notes in computer science on Advances in object-oriented database systems*, pages 334–339, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[Mei07] Erik Meijer. Confessions of a used programming language salesman. *SIGPLAN Not.*, 42(10):677–694, 2007.

[Mic04] Sun Microsystems. JSR 175: A metadata facility for the JavaTM programming language, September 2004.

[Mic05] Microsoft. The LINQ project: .NET language integrated query. White paper, September 2005.

[Mic06a] Microsoft. DLinq: .NET language integrated query for relational data, May 2006.

[Mic06b] Microsoft. Xlinq overview, May 2006. `http://msdn.microsoft.com/data/ref/linq/`.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MK05] Russell A. McClure and Ingolf H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 88–96, 2005.

[ML01] David Maier and Ming-Ju Lee. Object-oriented database theory: an introduction & indexing in OODBS, 2001.

[Mor79a] R. Morrison. S-Algol language reference manual. Technical Report CS/79/1, University of St Andrews, 1979.

[Mor79b] Ronald Morrison. *On the Development of Algol*. PhD thesis, University of St Andrews, 1979.

[Mot] Markus Mottl. PostgreSQL-OCaml. `http://hg.ocaml.info/release/postgresql-ocaml`.

[MPY07] Kevin Marshall, Chad Pytel, and Jon Yurek. *Pro Active Record: Databases with Ruby and Rails*. Apress, September 2007.

[MR97] David Melski and Thomas Reps. Interconvertibility of set constraints and context-free language reachability. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 74–89, New York, NY, USA, 1997. ACM Press.

[MS01] Erik Meijer and Wolfram Schulte. XML types for C#. BillG ThinkWeek Submission, Winter 2001.

[MSB03a] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with circles, triangles and rectangles. In *In XML Conference and Exposition*, 2003.

[MSB03b] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying tables, objects and documents. In *In Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL*, 2003.

[MTY05] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 320–330, New York, USA, 2005. ACM.

[OB88] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 174–183, New York, NY, USA, 1988. ACM.

[OBBT89] Atsushi Ohori, Peter Buneman, and Val Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 46–57, New York, NY, USA, 1989. ACM Press.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[Pur06] Peter Purich. *Oracle TopLink Developer's Guide, 10g Release 3(10.1.3)*. Oracle Corporation, January 2006.

[RC93] Peter Rob and Carlos Coronel. *Database systems: design, implementation and management*. Wadsworth Publ. Co., Belmont, CA, USA, 1993.

[Sar97] Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997.

[Sun04] Sun Microsystems. *JDK 5.0 Developer's Guide: Annotations*, 2004.

[SV06] Alexandra Silva and Joost Visser. Functional pearl: Strong types for relational databases. In *ACM SIGPLAN 2006 Haskell Workshop*, New York, NY, USA, 2006. ACM.

[TCIK00] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000. Springer-Verlag.

[Tei07] Dario Teixeira. A brief introduction to PG'OCaml, September 2007.

[TK03] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167, New York, NY, USA, 2003. ACM.

[TVM+03] Sameer Tyagi, Michael Vorburger, Keiron McCammon, Heiko Bobzin, and Keiron McCannon. *Core Java Data Objects*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[TW89] P. Trinder and P. L. Wadler. List comprehensions and the relational calculus. In C. Hall, J. Hughes, and J. T. O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 187–202. Department of Computer Science, University of Glasgow, 1989.

[UA94] Susan Darling Urban and Taoufik Ben Abdellatif. An object-oriented query language interface to relational databases in a multidatabase database environment. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference*, pages 387–394, Jun 1994.

[VP95] Murali Venkatrao and Michael Pizzo. SQL/CLI – a new binding style for SQL. *SIGMOD Rec.*, 24(4):72–77, 1995.

[Wad90] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[WCK06] Meng Wang, Kung Chen, and Siau-Cheng Khoo. Type-directed weaving of aspects for higher-order functional languages. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 78–87, New York, USA, 2006. ACM.

[WNR02] Brian Wright, Janice Nygard, and Ekkehard Rohwedder. *Oracle9i SQLJ Developer's Guide and Reference, Release 2 (9.2)*, March 2002. Part No. A96655-01.

[Won93] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 26–36, New York, NY, USA, 1993. ACM.

[Won00] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.

[YZM+95] Clement T. Yu, Yi Zhang, Weiyi Meng, Won Kim, Gaoming Wang, Tracy Pham, and Son Dao. Translation of object-oriented queries to relational queries. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 90–97, Washington, DC, USA, 1995. IEEE Computer Society.