Optimizing Database-Backed Applications with Query Synthesis

Alvin Cheung Armando Solar-Lezama Samuel Madden *
MIT CSAIL
{akcheung, asolar, madden}@csail.mit.edu

Abstract

Object-relational mapping libraries are a popular way for applications to interact with databases because they provide transparent access to the database using the same language as the application. Unfortunately, using such frameworks often leads to poor performance, as modularity concerns encourage developers to implement relational operations in application code. Such application code does not take advantage of the optimized relational implementations that database systems provide, such as efficient implementations of joins or push down of selection predicates.

In this paper we present QBS, a system that automatically transforms fragments of application logic into SQL queries. QBS differs from traditional compiler optimizations as it relies on synthesis technology to generate invariants and postconditions for a code fragment. The postconditions and invariants are expressed using a new theory of ordered relations that allows us to reason precisely about both the contents and order of the records produced complex code fragments that compute joins and aggregates. The theory is close in expressiveness to SQL, so the synthesized postconditions can be readily translated to SQL queries. Using 75 code fragments automatically extracted from over 120k lines of open-source code written using the Java Hibernate ORM, we demonstrate that our approach can convert a variety of imperative constructs into relational specifications and significantly improve application performance asymptotically by orders of magnitude.

Categories and Subject Descriptors. D.3.2 [*Programming Languages*]: Processors—Compilers; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis
Keywords. program optimization; program synthesis; performance

1. Introduction

In this paper, we develop QBS (Query By Synthesis), a new code analysis algorithm designed to make database-backed applications more efficient. Specifically, QBS identifies places where application logic can be pushed into the SQL queries issued by the application, and automatically transforms the code to do this. Moving application code into the database reduces the amount of data sent from the database to the application, and it also allows the database query optimizer to choose more efficient implementations of some operations—for instance, using indices to evaluate predicates or selecting efficient join algorithms.

* The authors are grateful for the support of Intel Corporation and NSF Grants 1065219 and 1139056.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

One specific target of QBS is programs that interact with the database through object-relational mapping layers (ORMs) such as Hibernate for Java. Our optimizations are particularly important for such programs because ORM layers often lead programmers to write code that iterates over collections of database records, performing operations like filters and joins that could be better done inside of the database. Such ORM layers are becoming increasingly popular; for example, as of March, 2013, on the job board `dice.com` 15% of the 17,000 Java developer jobs are for programmers with Hibernate experience.

We are not the first researchers to address this problem; Wiedermann *et al.* [37, 38] identified this as the *query extraction problem*. However, our work is able analyze a significantly larger class of source programs and generate a more expressive set of SQL queries than this prior work. Specifically, to the best of our knowledge, our work is the first that is able to identify joins and aggregates in general purpose application logic and convert them to SQL queries. Our analysis ensures that the generated queries are *precise* in that both the contents and the order of records in the generated queries are the same as those produced by the original code.

At a more foundational level, our paper is the first to demonstrate the use of constraint-based synthesis technology to attack a challenging compiler optimization problem. Our approach builds on the observation by Iu *et al.* [20] that if we can express the postcondition for an imperative code block in relational algebra, then we can translate that code block into SQL. Our approach uses constraint-based synthesis to automatically derive loop invariants and postconditions, and then uses an SMT solver to check the resulting verification conditions. In order to make synthesis and verification tractable, we define a new *theory of ordered relations* (TOR) that is close in expressiveness to SQL, while being expressive enough to concisely describe the loop invariants necessary to verify the codes of interest. The postconditions expressed in TOR can be readily translated to SQL, allowing them to be optimized by the database query planner and leading in some cases to orders of magnitude performance improvements.

In summary, this paper makes the following contributions:

1. We demonstrate a new approach to compiler optimization based on constraint-based synthesis of loop invariants and apply it to the problem of transforming low-level loop nests into high-level SQL queries.
2. We define a theory of ordered relations that allows us to concisely represent loop invariants and postconditions for code fragments that implement SQL queries, and to efficiently translate those postconditions into SQL.
3. We define a program analysis algorithm that identifies candidate code blocks that can potentially be transformed by QBS.
4. We demonstrate our full implementation of QBS and the candidate identification analysis for Java programs by automatically identifying and transforming 75 code fragments in two large open source projects. These transformations result in order-of-magnitude performance improvements. Although those projects

```
1  List<User> getRoleUser () {
2    List<User> listUsers = new ArrayList<User>();
3    List<User> users = this.userDao.getUsers();
4    List<Role> roles = this.roleDao.getRoles();
5    for (User u : users) {
6      for (Roles r : roles) {
7        if (u.roleId().equals(r.roleId())) {
8          User userok = u;
9          listUsers.add(userok);
10   }}}
11   return listUsers;
12 }
```

Figure 1: Sample code that implements join operation in application code, abridged from actual source for clarity

```
1  List listUsers := [ ]; int i, j = 0;
2  List users := Query(SELECT * FROM users);
3  List roles = Query(SELECT * FROM roles);
4  while (i < users.size()) {
5    while (j < roles.size()) {
6      if (users[i].roleId = roles[j].roleId)
7        listUsers := append(listUsers, users[i]);
8      ++j;
9    }
10   ++i;}
```

Figure 2: Sample code expressed in kernel language

Postcondition

$listUsers = \pi_{\ell}(\bowtie_{\varphi}(users, roles))$
where
$\varphi(e_{users}, e_{roles}) := e_{users}.roleId = e_{roles}.roleId$
ℓ contains all the fields from the *User* class

Translated code

```
1  List<User> getRoleUser () {
2    List<User> listUsers = db.executeQuery(
3          "SELECT u
4           FROM users u, roles r
5           WHERE u.roleId == r.roleId
6           ORDER BY u.roleId, r.roleId");
7    return listUsers; }
```

Figure 3: Postcondition as inferred from Fig. 1 and code after query transformation

use ORM libraries to retrieve persistent data, our analysis is not specific to ORM libraries and is applicable to programs with embedded SQL queries.

2. Overview

This section gives an overview of our compilation infrastructure and the QBS algorithm to translate imperative code fragments to SQL. We use as a running example a block of code extracted from an open source project management application [2] written using the Hibernate framework. The original code was distributed across several methods which our system automatically collapsed into a single continuous block of code as shown in Fig. 1. The code retrieves the list of users from the database and produces a list containing a subset of users with matching roles.

The example implements the desired functionality but performs poorly. Semantically, the code performs a relational join and projection. Unfortunately, due to the lack of global program information, the ORM library can only fetch all the users and roles from the database and perform the join in application code, without utilizing indices or efficient join algorithms the database system has access to. QBS fixes this problem by compiling the sample code to that

$$
\begin{array}{rcl}
c \in \text{constant} & ::= & \text{True} \mid \text{False} \mid \text{number literal} \mid \text{string literal} \\
e \in \text{expression} & ::= & c \mid [\,] \mid var \mid e.f \mid \{f_i = e_i\} \mid e_1\ op\ e_2 \mid \neg e \\
 & \mid & \text{Query}(\ldots) \mid \text{size}(e) \mid \text{get}_{e_r}(e_s) \\
 & \mid & \text{append}(e_r, e_s) \mid \text{unique}(e) \\
c \in \text{command} & ::= & \text{skip} \mid var := e \mid \text{if}(e)\ \text{then}\ c_1\ \text{else}\ c_2 \\
 & \mid & \text{while}(e)\ \text{do}\ c \mid c_1\ ;\ c_2 \mid \text{assert}\ e \\
op \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid =
\end{array}
$$

Figure 4: Abstract syntax of the kernel language

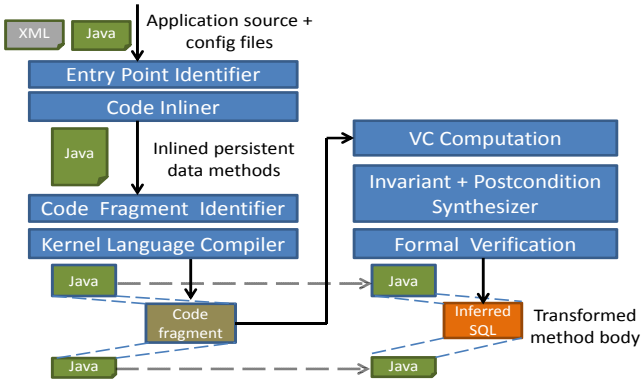


Figure 5: QBS architecture

shown at the bottom of Fig. 3. The nested loop is converted to an SQL query that implements the same functionality in the database where it can be executed more efficiently. Note that the query imposes an order on the retrieved records; this is because in general, nested loops can constraint the ordering of the output records in ways that need to be captured by the query.

In order to apply the QBS algorithm to perform the desired conversion, our system must be able to cope with the complexities of real-world Java code such as aliasing and method calls, which obscure opportunities for transformations. For example, it would not be possible to transform the code fragment in Fig. 1 without knowing that getUsers and getRoles execute specific queries on the database and return non-aliased lists of results, so the first step of the system is to identify promising code fragments and translate them into a simpler kernel language shown in Fig. 4.

The kernel language operates on three types of values: scalars, immutable records, and immutable lists. Lists represent the collections of records and are used to model the results that are returned from database retrieval operations. Lists store either scalar values or records constructed with scalars, and nested lists are assumed to be appropriately flattened. The language currently does not model the three-valued logic of null values in SQL, and does not model updates to the database. The semantics of the constructs in the kernel language are mostly standard, with a few new ones introduced for record retrievals. Query(...) retrieves records from the database and the results are returned as a list. The records of a list can be randomly accessed using get, and records can be appended to a list using append. Finally, unique takes in a list and creates a new list with all duplicate records removed. Fig. 2 shows the example translated to the kernel language.

2.1 QBS Architecture

We now discuss the architecture of QBS and describe the steps in inferring SQL queries from imperative code. The architecture of QBS is shown in Fig. 5.

Identify code fragments to transform. Given a web application written in Java, QBS first finds the persistent data methods in the application, which are those that fetch persistent data via ORM li-

brary calls. It also locates all entry points to the application such as servlet handlers. From each persistent data method that is reachable from the entry points, the system inlines a neighborhood of calls, *i.e.* a few of the parent methods that called the persistent data method and a few of the methods called by them. If there is ambiguity as to the target of a call, all potential targets are considered up to a budget. A series of analyses is then performed on each inlined method body to identify a continuous code fragment that can be potentially transformed to SQL; ruling out, for example, code fragments with side effects. For each candidate code fragment, our system automatically detects the program variable that will contain the results from the inferred query (in the case of the running example it is listUsers) — we refer this as the "result variable." At the end of the process, each code fragment is converted to our kernel language as discussed.

Compute verification conditions. As the next step, the system computes the verification conditions of the code fragment expressed in the kernel language. The verification conditions are written using the predicate language derived from the theory of ordered relations to be discussed in Sec. 3. The procedure used to compute verification conditions is a fairly standard one [12, 16]; the only twist is that the verification condition must be computed in terms of an unknown postcondition and loop invariants. The process of computing verification conditions is discussed in more detail in Sec. 4.1.

Synthesize loop invariants and postconditions. The definitions of the postcondition and invariants need to be filled in and validated before translation can proceed. QBS does this using a synthesis-based approach that is similar to [31], where a synthesizer is used to come up with a postcondition and invariants that satisfy the computed verification conditions. The synthesizer uses a symbolic representation of the space of candidate postconditions and invariants and efficiently identifies candidates within the space that are correct according to a bounded verification procedure. It then uses a theorem prover (Z3 [3], specifically) to check if those candidates can be proven correct. The space of candidate invariants and postconditions is described by a template generated automatically by the compiler. To prevent the synthesizer from generating trivial postconditions (such as True), the template limits the synthesizer to only generate postconditions that can be translated to SQL as defined by our theory of ordered relations, such as that shown at the top of Fig. 3.

We observe that it is not necessary to determine the strongest (in terms of logical implication) invariants or postconditions: we are only interested in finding postconditions that allow us transform the input code fragment into SQL. In the case of the running example, we are only interested in finding a postcondition of the form listUsers = *query*, where *query* is an expression translatable to SQL. Similarly, we only need to discover loop invariants that are strong enough to prove the postcondition of interest. From the example shown in Fig. 1, our system infers the postcondition shown at the top of Fig. 3, where π, σ, and $\bowtie$ are ordered versions of relational projection, selection, and join, respectively to be defined in Sec. 3. The process of automatic template generation from the input code fragment and synthesis of the postcondition from the template are discussed in Sec. 4.

Unfortunately, determining loop invariants is undecidable for arbitrary programs [6], so there will be programs for which the necessary invariants fall outside the space defined by the templates. However, our system is significantly more expressive than the state of the art as demonstrated by our experiments in Sec. 7.

Convert to SQL. After the theorem prover verifies that the computed invariants and postcondition are correct, the input code fragment is translated to SQL, as shown in the bottom of Fig. 3. The predicate language defines syntax-driven rules to translate any expressions in the language into valid SQL. The details of validation is discussed in Sec. 5 while the rules for SQL conversion are introduced in Sec. 3.2. The converted SQL queries are patched back into the original code fragments and compiled as Java code.

$$
\begin{array}{rcl}
c \in \text{constant} & ::= & \mathsf{True} \mid \mathsf{False} \mid \text{number literal} \mid \text{string literal} \\
e \in \text{expression} & ::= & c \mid [\,] \mid \text{program var} \mid \{f_i = e_i\} \mid e_1\ op\ e_2 \mid \neg\, e \\
& \mid & \mathsf{Query}(\ldots) \mid \mathsf{size}(e) \mid \mathsf{get}_{e_s}(e_r) \mid \mathsf{top}_{e_s}(e_r) \\
& \mid & \pi_{[f_{i_1},\ldots,f_{i_N}]}(e) \mid \sigma_{\varphi\sigma}(e) \mid \bowtie_{\varphi\bowtie}(e_1, e_2) \\
& \mid & \mathsf{sum}(e) \mid \mathsf{max}(e) \mid \mathsf{min}(e) \\
& \mid & \mathsf{append}(e_r, e_s) \mid \mathsf{sort}_{[f_{i_1},\ldots,f_{i_N}]}(e) \mid \mathsf{unique}(e) \\
op \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid = \\
\varphi_\sigma \in \text{select func} & ::= & p_{\sigma_1} \wedge \ldots \wedge p_{\sigma_N} \\
p_\sigma \in \text{select pred} & ::= & e.f_i\ op\ c \mid e.f_i\ op\ e.f_j \mid \mathsf{contains}(e, e_r) \\
\varphi_\bowtie \in \text{join func} & ::= & p_{\bowtie_1} \wedge \ldots \wedge p_{\bowtie_N} \\
p_\bowtie \in \text{join pred} & ::= & e_1.f_i\ op\ e_2.f_j
\end{array}
$$

Figure 6: Abstract syntax for the predicate language based on the theory of ordered relations

3. Theory of Finite Ordered Relations

QBS uses a theory of finite ordered relations to describe postconditions and invariants. The theory is defined to satisfy four main requirements: precision, expressiveness, conciseness and ease of translation to SQL. For precision, we want to be able to reason about *both* the contents and order of records retrieved from the database. This is important because in the presence of joins, the order of the result list will not be arbitrary even when the original list was arbitrary, and we do not know what assumptions the rest of the program makes on the order of records. The theory must also be expressive enough not just to express queries but also to express invariants, which must often refer to partially constructed lists. For instance, the loop invariants for the sample code fragment in Fig. 1 must express the fact that listUsers is computed from the first i and j records of users and roles respectively. Conciseness, *e.g.*, the number of relational operators involved, is important because the complexity of synthesis grows with the size of the synthesized expressions, so if we can express invariants succinctly, we will be able to synthesize them more efficiently. Finally, the inferred postconditions must be translatable to standard SQL.

There are many ways to model relational operations (see Sec. 8), but we are not aware of any that fulfills all of the criteria above. For example, relational algebra is not expressive enough to describe sufficiently precise loop invariants. Defined in terms of sets, relational algebra cannot naturally express concepts such as "the first i elements of the list." First order logic (FOL), on the other hand, is very expressive, but it would be hard to translate arbitrary FOL expressions into SQL.

3.1 Basics

Our theory of finite ordered relations is essentially relational algebra defined in terms of lists instead of sets. The theory operates on three types of values: scalars, records, and ordered relations of finite length. Records are collections of named fields, and an ordered relation is a finite list of records. Each record in the relation is labeled with an integer index that can be used to fetch the record. Figure 6 presents the abstract syntax of the theory and shows how to combine operators to form expressions.

The semantics of the operators in the theory are defined recursively by a set of axioms; a sample of which are shown in Fig. 7. get and top take in an ordered relation e_r and return the record stored at index e_s or all the records from index 0 up to index e_s respec-

top

$$\frac{r = [\,]}{\mathsf{top}_r(i) = [\,]} \quad \frac{i = 0}{\mathsf{top}_r(i) = [\,]} \quad \frac{i > 0 \quad r = h : t}{\mathsf{top}_r(i) = h : \mathsf{top}_t(i-1)}$$

join ($\bowtie$)

$$\frac{r_1 = [\,]}{\bowtie_\varphi(r_1, r_2) = [\,]} \quad \frac{r_2 = [\,]}{\bowtie_\varphi(r_1, r_2) = [\,]} \quad \frac{r_1 = h : t}{\bowtie_\varphi(r_1, r_2) = \mathsf{cat}(\bowtie'_\varphi(h, r_2), \bowtie_\varphi(t, r_2))}$$

$$\frac{r_2 = h : t \quad \varphi(e, h) = \mathsf{True}}{\bowtie'_\varphi(e, r_2) = (e, h) : \bowtie'_\varphi(e, t)} \quad \frac{r_2 = h : t \quad \varphi(e, h) = \mathsf{False}}{\bowtie'_\varphi(e, r_2) = \bowtie'_\varphi(e, t)}$$

projection (π)

$$\frac{r = [\,]}{\pi_\ell(r) = [\,]} \quad \frac{r = h : t \quad f_i \in \ell \quad h.f_i = e_i}{\pi_\ell(r) = \{f_i = e_i\} : \pi_\ell(t)}$$

selection (σ)

$$\frac{r = [\,]}{\sigma_\varphi(r) = [\,]} \quad \frac{r = h : t \quad \varphi(h) = \mathsf{True}}{\sigma_\varphi(r) = h : \sigma_\varphi(t)} \quad \frac{r = h : t \quad \varphi(h) = \mathsf{False}}{\sigma_\varphi(r) = \sigma_\varphi(t)}$$

max

$$\frac{r = [\,]}{\mathsf{max}(r) = -\infty} \quad \frac{r = h : t \quad h > \mathsf{max}(t)}{\mathsf{max}(r) = h} \quad \frac{r = h : t \quad h \leq \mathsf{max}(t)}{\mathsf{max}(r) = \mathsf{max}(t)}$$

Figure 7: Some of the axioms that define the theory of ordered relations, Appendix C contains the full list of axioms

tively. The definitions for π, σ and $\bowtie$ are modeled after relational projection, selection, and join respectively, but they also define an order for the records in the output relation relative to those in the input relations. The projection operator π creates new copies of each record, except that for each record only those fields listed in $[f_{i_1}, \ldots, f_{i_N}]$ are retained. Like projection in relational algebra, the same field can be replicated multiple times. The σ operator uses a selection function φ_σ to filter records from the input relation. φ_σ is defined as a conjunction of predicates, where each predicate can compare the value of a record field and a constant, the values of two record fields, or check if the record is contained in another relation e_r using contains. Records are added to the resulting relation if the function returns True. The $\bowtie$ operator iterates over each record from the first relation and pairs it with each record from the second relation. The two records are passed to the join function $\varphi_\bowtie$. Join functions are similar to selection functions, except that predicates in join functions compare the values of the fields from the input ordered relations. The axioms that define the aggregate operators max, min, and sum assume that the input relation contains only one numeric field, namely the field to aggregate upon.

The definitions of unique and sort are standard; in the case of sort, $[f_{i_1}, \ldots, f_{i_N}]$ contains the list of fields to sort the relation by. QBS does not actually reason about these two operations in terms of their definitions; instead it treats them as uninterpreted functions with a few algebraic properties, such as

$$\bowtie_\varphi(\mathsf{sort}_{\ell_1}(r_1), \mathsf{sort}_{\ell_2}(r_2)) = \mathsf{sort}_{\mathsf{cat}(\ell_1,\ell_2)}(\bowtie_\varphi(r_1, r_2)).$$

Because of this, there are some formulas involving sort and unique that we cannot prove, but we have not found this to be significant in practice (see Sec. 7 for details).

3.2 Translating to SQL

The expressions defined in the predicate grammar can be converted into semantically equivalent SQL queries. In this section we prove that any expression that does not use append or unique can be compiled into an equivalent SQL query. We prove this in three steps; first, we define base and sorted expressions, which are formulated based on SQL expressions without and with ORDER BY clauses respectively. Next, we define *translatable expressions* and show that any expression that does not use append or unique can be converted into a translatable expression. Then we show how to produce SQL from translatable expressions.

Definition 1 (**Translatable Expressions).** *Any* transExp *as defined below can be translated into SQL:*

$$\begin{array}{rcl} b \in \mathsf{baseExp} & ::= & \mathsf{Query}(\ldots) \mid \mathsf{top}_e(s) \mid \bowtie_{\mathsf{True}}(b_1, b_2) \mid \mathsf{agg}(t) \\ s \in \mathsf{sortedExp} & ::= & \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))) \\ t \in \mathsf{transExp} & ::= & s \mid \mathsf{top}_e(s) \end{array}$$

where the term agg *in the grammar denotes any of the aggregation operators (*min, max, sum, size*).*

$$\begin{array}{rcl} [\![\mathsf{Query}(\mathit{string})]\!] & = & (\ \mathit{string}\ ) \\ [\![\mathsf{top}_e(s)]\!] & = & \mathsf{SELECT}\ *\ \mathsf{FROM}\ [\![s]\!]\ \mathsf{LIMIT}\ [\![e]\!] \\ [\![\bowtie_{\mathsf{True}}(t_1, t_2)]\!] & = & \mathsf{SELECT}\ *\ \mathsf{FROM}\ [\![t_1]\!], [\![t_2]\!] \\ [\![\mathsf{agg}(t)]\!] & = & \mathsf{SELECT}\ \mathsf{agg}(\mathit{field})\ \mathsf{FROM}\ [\![t]\!] \\ [\![\pi_{\ell_1}(\mathsf{sort}_{\ell_2}(\sigma_{\varphi_\sigma}(t)))]\!] & = & \mathsf{SELECT}\ [\![\ell_1]\!]\ \mathsf{FROM}\ [\![t]\!]\ \mathsf{WHERE}\ [\![\varphi_\sigma]\!] \\ & & \mathsf{ORDER\ BY}\ [\![\ell_2]\!], \mathsf{Order}(t) \\ [\![\mathsf{unique}(t)]\!] & = & \mathsf{SELECT\ DISTINCT}\ *\ \mathsf{FROM}\ [\![t]\!] \\ & & \mathsf{ORDER\ BY}\ \mathsf{Order}(t) \\ [\![\varphi_\sigma(e)]\!] & = & [\![e]\!].f_1\ \mathit{op}\ [\![e]\!]\ \mathsf{AND}\ \ldots\ \mathsf{AND}\ [\![e]\!].f_N\ \mathit{op}\ [\![e]\!] \\ [\![\mathsf{contains}(e, t)]\!] & = & [\![e]\!]\ \mathsf{IN}\ [\![t]\!] \\ [\![[f_{i_1}, \ldots, f_{i_N}]]\!] & = & f_{i_1}, \ldots, f_{i_N} \end{array}$$

Figure 8: Syntactic rules to convert translatable expressions to SQL

Theorem 1 (**Completeness of Translation Rules).** *All expressions in the predicate grammar in Fig. 6, except for those that contain* append *or* unique*, can be converted into translatable expressions.*

The theorem is proved by defining a function Trans that maps any expression to a translatable expression and showing that the mapping is semantics preserving. Definition of Trans is shown in Appendix B. Semantic equivalence between the original and the translated expression is proved using the expression equivalences listed in Thm. 2. Using those equivalences, for example, we can show that for $s \in \mathsf{sortedExp}$ and $b \in \mathsf{baseExp}$:

$$\begin{array}{rll} \mathsf{Trans}(\sigma_{\varphi'_\sigma}(s)) & = \mathsf{Trans}(\sigma_{\varphi'}(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))))) & [\mathsf{sortedExp\ def.}] \\ & = \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma \wedge \varphi'_\sigma}(b))) & [\mathsf{Trans\ def.}] \\ & = \pi_{\ell_\pi}(\sigma_{\varphi'_\sigma}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma}(b)))) & [\mathsf{expression\ equiv.}] \\ & = \sigma_{\varphi'_\sigma}(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma}(b)))) & [\mathsf{expression\ equiv.}] \\ & = \sigma_{\varphi'_\sigma}(s) & [\mathsf{sortedExp\ def.}] \end{array}$$

Thus the semantics of the original TOR expression is preserved.

Translatable expressions to SQL. Following the syntax-directed rules in Fig. 8, any translatable expression can be converted into an equivalent SQL expression. Most rules in Fig. 8 are direct translations from the operators in the theory into their SQL equivalents.

One important aspect of the translation is the way that ordering of records is preserved. Ordering is problematic because although the operators in the theory define the order of the output in terms of the order of their inputs, SQL queries are not guaranteed to preserve the order of records from nested sub-queries; *e.g.*, the ordering imposed by an ORDER BY clause in a nested query is not guaranteed to be respected by an outer query that does not impose any ordering on the records.

To solve this problem, the translation rules introduce a function Order—defined in Fig. 9—which scans a translatable expression t and returns a list of fields that are used to order the subexpressions

$$\mathsf{Order}(\mathsf{Query}(...)) = [\text{record order in DB}] \quad \mathsf{Order}(\mathsf{agg}(e)) = [\,]$$
$$\mathsf{Order}(\mathsf{top}_i(e)) = \mathsf{Order}(e) \quad \mathsf{Order}(\mathsf{unique}(e)) = \mathsf{Order}(e)$$
$$\mathsf{Order}(\pi_\ell(e)) = \mathsf{Order}(e) \quad \mathsf{Order}(\sigma_\varphi(e)) = \mathsf{Order}(e)$$
$$\mathsf{Order}(\bowtie_\varphi(e_1, e_2)) = \mathsf{cat}(\mathsf{Order}(e_1), \mathsf{Order}(e_2))$$
$$\mathsf{Order}(\mathsf{sort}_\ell(e)) = \mathsf{cat}(\ell, \mathsf{Order}(e))$$

Figure 9: Definition of Order

in t. The list is then used to impose an ordering on the outer SQL query with an ORDER BY clause. One detail of the algorithm not shown in the figure is that some projections in the inner queries need to be modified so they do not eliminate fields that will be needed by the outer ORDER BY clause, and that we assume Query(...) is ordered by the order in which the records are stored in the database (unless the query expression already includes an ORDER BY clause).

Append and Unique. The append operation is not included in translatable expressions because there is no simple means to combine two relations in SQL that preserves the ordering of records in the resulting relation [1]. We can still translate unique, however, using the SELECT DISTINCT construct at the outermost level, as Fig. 8 shows. Using unique in nested expressions, however, can change the semantics of the results in ways that are difficult to reason about (*e.g.*, $\mathsf{unique}(\mathsf{top}_e(r))$ is not equivalent to $\mathsf{top}_e(\mathsf{unique}(r))$). Thus, the only expressions with unique that we translate to SQL are those that use it at the outermost level. In our experiments, we found that omitting those two operators did not significantly limit the expressiveness of the theory.

With the theory in mind, we now turn to the process of computing verification conditions of the input code fragments.

4. Synthesis of Invariants and Postconditions

Given an input code fragment in the kernel language, the next step in QBS is to come up with an expression for the result variable of the form $\mathit{resultVar} = e$, where e is a translatable expression as defined in Sec. 3.2.

4.1 Computing Verification Conditions

In order to infer the postcondition, we compute *verification conditions* for the input code fragment using standard techniques from axiomatic semantics [19]. As in traditional Hoare style verification, computing the verification condition of the while statements involves a loop invariant. Unlike traditional computation of verification conditions, however, both the postcondition and the loop invariants are unknown when the conditions are generated. This does not pose problems for QBS as we simply treat invariants (and the postcondition) as unknown predicates over the program variables that are currently in scope when the loop is entered.

As an example, Fig. 11 shows the verification conditions that are generated for the running example. In this case, the verification conditions are split into two parts, with invariants defined for both loops.

The first two assertions describe the behavior of the outer loop on line 5, with the first one asserting that the outer loop invariant must be true on entry of the loop (after applying the rule for the assignments prior to loop entry), and the second one asserting that the postcondition for the loop is true when the loop terminates. The third assertion asserts that the inner loop invariant is true when it is first entered, given that the outer loop condition and loop invariant

[1] One way to preserve record ordering in list append is to use case expressions in SQL, although some database systems such as SQL Server limit the number of nested case expressions.

Verification conditions for the outer loop (oInv = outerLoopInvariant, iInv = innerLoopInvariant, pcon = postCondition)	
initialization	$\mathsf{oInv}(0, \mathit{users}, \mathit{roles}, [\,])$
loop exit	$i \geq \mathsf{size}(\mathit{users}) \wedge \mathsf{oInv}(i, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow \mathsf{pcon}(\mathit{listUsers}, \mathit{users}, \mathit{roles})$
perservation	(same as inner loop initialization)
Verification conditions for the inner loop	
initialization	$i < \mathsf{size}(\mathit{users}) \wedge \mathsf{oInv}(i, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow \mathsf{iInv}(i, 0, \mathit{users}, \mathit{roles}, \mathit{listUsers})$
loop exit	$j \geq \mathsf{size}(\mathit{roles}) \wedge \mathsf{iInv}(i, j, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow \mathsf{oInv}(i+1, \mathit{users}, \mathit{roles}, \mathit{listUsers})$
preservation	$j < \mathsf{size}(\mathit{roles}) \wedge \mathsf{iInv}(i, j, \mathit{users}, \mathit{roles}, \mathit{listUsers}) \rightarrow (\mathsf{get}_i(\mathit{users}).\mathit{id} = \mathsf{get}_j(\mathit{roles}).\mathit{id} \wedge \mathsf{iInv}(i, j+1, \mathit{users}, \mathit{roles}, \mathsf{append}(\mathit{listUsers}, \mathsf{get}_i(\mathit{users})))) \vee (\mathsf{get}_i(\mathit{users}).\mathit{id} \neq \mathsf{get}_j(\mathit{roles}).\mathit{id} \wedge \mathsf{iInv}(i, j+1, \mathit{users}, \mathit{roles}, \mathit{listUsers}))$

Figure 11: Verification conditions for the running example

are true. The preservation assertion is the inductive argument that the inner loop invariant is preserved after executing one iteration of the loop body. The list *listUsers* is either appended with a record from $\mathsf{get}_i(\mathit{users})$, or remains unchanged, depending on whether the condition for the if statement, $\mathsf{get}_i(\mathit{users}).\mathit{id} = \mathsf{get}_j(\mathit{roles}).\mathit{id}$, is true or not. Finally, the loop exit assertion states that the outer loop invariant is valid when the inner loop terminates.

4.2 Constraint based synthesis

The goal of the synthesis step is to derive postcondition and loop invariants that satisfy the verification conditions generated in the previous step. We synthesize these predicates using the SKETCH constraint-based synthesis system [30]. In general, SKETCH takes as input a program with "holes" and uses a counterexample guided synthesis algorithm (CEGIS) to efficiently search the space of all possible completions to the holes for one that is correct according to a bounded model checking procedure. For QBS, the program is a simple procedure that asserts that the verification conditions hold for all possible values of the free variables within a certain bound. For each of the unknown predicates, the synthesizer is given a *sketch* (*i.e.*, a template) that defines a space of possible predicates which the synthesizer will search. The sketches are automatically generated by QBS from the kernel language representation.

4.3 Inferring the Space of Possible Invariants

Recall that each invariant is parameterized by the current program variables that are in scope. Our system assumes that each loop invariant is a conjunction of predicates, with each predicate having the form $\mathit{lv} = e$, where lv is a program variable that is modified within the loop, and e is an expression in TOR.

The space of expressions e is restricted to expressions of the same static type as lv involving the variables that are in scope. The system limits the size of expressions that the synthesizer can consider, and incrementally increases this limit if the synthesizer fails to find any candidate solutions (to be explained in Sec. 4.5).

Fig. 10 shows a stylized representation of the set of predicates that our system considers for the outer loop in the running example. The figure shows the potential expressions for the program variable i and *listUsers*. One advantage of using the theory of ordered relations is that invariants can be relatively concise. This has a big impact for synthesis, because the space of expressions grows exponentially with respect to the size of the candidate expressions.

4.4 Creating Templates for Postconditions

The mechanism used to generate possible expressions for the result variable is similar to that for invariants, but we have stronger

$$i \; op \left\{ \begin{array}{c} i \mid \mathsf{size}(users) \mid \mathsf{size}(roles) \mid \mathsf{size}(listUsers) \mid \\ \mathsf{sum}(\pi_\ell(users) \mid \mathsf{sum}(\pi_\ell(roles) \mid \mathsf{max}(\pi_\ell(users) \mid \\ \text{[other relational expressions that return a scalar value]} \end{array} \right\} \wedge listUsers = \left\{ \begin{array}{c} listUsers \mid \sigma_\varphi(users) \mid \\ \pi_\ell(\bowtie_\varphi(\mathsf{top}_{e_1}(users), \mathsf{top}_{e_2}(roles))) \mid \\ \pi_\ell(\bowtie_{\varphi_3}(\sigma_{\varphi_1}(\mathsf{top}_{e_1}(users), \sigma_{\varphi_2}(\mathsf{top}_{e_2}(roles))))) \mid \\ \text{[other relational expressions that return an ordered list]} \end{array} \right\}$$

Figure 10: Space of possible invariants for the outer loop of the running example.

restrictions, since we know the postcondition must be of the form $resultVar = e$ in order to be translatable to SQL.

For the running example, QBS considers the following possible set of postconditions:

$$listUsers = \left\{ \begin{array}{c} users \mid \sigma_\varphi(users) \mid \mathsf{top}_e(users) \mid \\ \pi_\ell(\bowtie_\varphi(\mathsf{top}_{e_1}(users), \mathsf{top}_{e_2}(roles))) \mid \\ \pi_\ell(\bowtie_{\varphi_3}(\sigma_{\varphi_1}(\mathsf{top}_{e_1}(users), \sigma_{\varphi_2}(\mathsf{top}_{e_2}(roles))))) \mid \\ \text{[other relational expressions that return an ordered list]} \end{array} \right\}$$

4.5 Optimizations

The basic algorithm presented above for generating invariant and postcondition templates is sufficient but not efficient for synthesis. In this section we describe two optimizations that improve the synthesis efficiency.

Incremental solving. As an optimization, the generation of templates for invariants and postconditions is done in an iterative manner: QBS initially scans the input code fragment for specific patterns and creates simple templates using the production rules from the predicate grammar, such as considering expressions with only one relational operator, and functions that contains one boolean clause. If the synthesizer is able to generate a candidate that can be used to prove the validity of the verification conditions, then our job is done. Otherwise, the system repeats the template generation process, but increases the complexity of the template that is generated by considering expressions consisting of more relational operators, and more complicated boolean functions. Our evaluation using real-world examples shows that most code examples require only a few (< 3) iterations before finding a candidate solution. Additionally, the incremental solving process can be run in parallel.

Breaking symmetries. Symmetries have been shown to be one of sources of inefficiency in constraint solvers [11, 34]. Unfortunately, the template generation algorithm presented above can generate highly symmetrical expressions. For instance, it can generate the following potential candidates for the postcondition:

$$\sigma_{\varphi_2}(\sigma_{\varphi_1}(users))$$
$$\sigma_{\varphi_1}(\sigma_{\varphi_2}(users))$$

Notice that the two expressions are semantically equivalent to the expression $\sigma_{\varphi_1 \wedge \varphi_2}(users)$. These are the kind of symmetries that are known to affect solution time dramatically. The template generation algorithm leverages known algebraic relationships between expressions to reduce the search space of possible expressions. For example, our algebraic relationships tell us that it is unnecessary to consider expressions with nested σ like the ones above. Also, when generating templates for postconditions, we only need to consider translatable expressions as defined in Sec. 3.2 as potential candidates. Our experiments have shown that applying these symmetric breaking optimizations can reduce the amount of solving time by half.

Even with these optimizations, the spaces of invariants considered are still astronomically large; on the order of 2^{300} possible combinations of invariants and postconditions for some problems. Thanks to these optimizations, however, the spaces can be searched very efficiently by the constraint based synthesis procedure.

Type	Expression inferred
outer loop invariant	$i \leq \mathsf{size}(users) \wedge$ $listUsers = \pi_\ell(\bowtie_\varphi(\mathsf{top}_i(users), roles))$
inner loop invariant	$i < \mathsf{size}(users) \wedge j \leq \mathsf{size}(roles) \wedge$ $listUsers = \mathsf{append}($ $\pi_\ell(\bowtie_\varphi(\mathsf{top}_i(users), roles)),$ $\pi_\ell(\bowtie_\varphi(\mathsf{get}_i(users), \mathsf{top}_j(roles)))$
postcondition	$listUsers = \pi_\ell(\bowtie_\varphi(users, roles))$

where $\varphi(e_{users}, e_{roles}) := e_{users}.roleId = e_{roles}.roleId$,
ℓ contains all the fields from the *User* class

Figure 12: Inferred expressions for the running example

5. Formal Validation and Source Transformation

After the synthesizer comes up with candidate invariants and postconditions, they need to be validated using a theorem prover, since the synthesizer used in our prototype is only able to perform bounded reasoning as discussed earlier. We have implemented the theory of ordered relations in the Z3 [3] prover for this purpose. Since the theory of lists is not decidable as it uses universal quantifiers, the theory of ordered relations is not decidable as well. However, for practical purposes we have not found that to be limiting in our experiments. In fact, given the appropriate invariants and postconditions, the prover is able to validate them within seconds by making use of the axioms that are provided.

If the prover can establish the validity of the invariants and postcondition candidates, the postcondition is then converted into SQL according to the rules discussed in Sec. 3.2. For instance, for the running example our algorithm found the invariants and postcondition as shown in Fig. 12, and the input code is transformed into the results in Fig. 3.

If the prover is unable to establish validity of the candidates (detected via a timeout), we ask the synthesizer to generate other candidate invariants and postconditions after increasing the space of possible solutions as described in Sec. 4.5. One reason that the prover may not be able to establish validity is because the maximum size of the relations set for the synthesizer was not large enough. For instance, if the code returns the first 100 elements from the relation but the synthesizer only considers relations up to size 10, then it will incorrectly generate candidates that claim that the code was performing a full selection of the entire relation. In such cases our algorithm will repeat the synthesis process after increasing the maximum relation size.

5.1 Incorporating inferred queries into original code

After verifying the transformation, the inferred queries are merged back into the code fragment. Before replacing the original fragment, we run a def-use analysis (which uses the points-to information to be described in Sec. 6.2) starting at the beginning of the code fragment until the end of the inlined method body. This is to ensure that none of the variables that are defined in the code fragment to be replaced is used in the rest of the inlined method after replacement.

6. Preprocessing of Input Programs

In order to handle real-world Java programs, QBS performs a number of initial passes to identify the code fragments to be transformed to kernel language representation before query inference. The code

identification process makes use of several standard analysis techniques, and in this section we describe them in detail.

6.1 Generating initial code fragments

As discussed in Sec. 2, code identification first involves locating application entry point methods and data persistent methods. From each data persistent method, our system currently inlines a neighborhood of 5 callers and callees. We only inline callees that are defined in the application, and provide models for native Java API calls. For callers we only inline those that can be potentially invoked from an entry point method. The inlined method bodies are passed to the next step of the process. Inlining improves the precision of the points-to information for our analysis. While there are other algorithms that can be used to obtain such information [36, 40], we chose inlining for ease of implementation and is sufficient in processing the code fragments used in the experiments.

6.2 Identifying code fragments for query inference

Given a candidate inlined method for query inference, we would like to identify the code fragment to transform to our kernel language representation. While we can simply use the entire body of the inlined method for this purpose, we would like to limit the amount of code to be analyzed, since including code that does not manipulate persistent data will increase the difficulty in synthesizing invariants and postconditions with no actual benefit. We accomplish this goal using a series of analyses. First, we run a flow-sensitive pointer analysis [27] on the body of the inlined method. The results of this analysis is a set of points-to graphs that map each reference variable to one or more abstract memory locations at each program point. Using the points-to information, we perform two further analyses on the inlined method.

Location tainting. We run a dataflow analysis that conservatively marks values that are derived from persistent data retrieved via ORM library calls. This analysis is similar to taint analysis [35], and the obtained information allows the system to remove regions of code that do not manipulate persistent data and thus can be ignored for our purpose. For instance, all reference variables and list contents in Fig. 1 will be tainted as they are derived from persistent data.

Value escapement. After that, we perform another dataflow analysis to check if any abstract memory locations are reachable from references that are outside of the inlined method body. This analysis is needed because if an abstract memory location `m` is accessible from the external environment (*e.g.*, via a global variable) after program point `p`, then converting `m` might break the semantics of the original code, as there can be external references to `m` that rely on the contents of `m` before the conversion. This analysis is similar to classical escape analysis [36]. Specifically, we define an abstract memory location `m` as having escaped at program point `p` if any of the following is true:

- It is returned from the entry point method.
- It is assigned to a global variable that persists after the entry point method returns (in the web application context, these can be variables that maintain session state, for instance).
- It is assigned to a `Runnable` object, meaning that it can be accessed by other threads.
- It is passed in as a parameter into the entry point method.
- It can be transitively reached from an escaped location.

With that in mind, we define the beginning of the code fragment to pass to the QBS algorithm as the program point `p` in the inlined method where tainted data is first retrieved from the database, and the end as the program point `p'` where tainted data first escapes, where `p'` appears after `p` in terms of control flow. For instance, in Fig. 1 the `return` statement marks the end of the code fragment, with the result variable being the value returned.

App	# persistent data code fragments	translated	rejected	failed
Wilos	33	21	9	3
itracker	16	12	0	4
Total	**49**	**33**	**9**	**7**

Figure 13: Real-world code fragments experiment

6.3 Compilation to kernel language

Each code fragment that is identified by the previous analysis is compiled to our kernel language. Since the kernel language is based on value semantics and does not model heap updates for lists, during the compilation process we translate list references to the abstract memory locations that they point to, using the results from earlier analysis. In general, there are cases where the preprocessing step fails to identify a code fragment from an inlined method (*e.g.*, persistent data values escape to multiple result variables under different branches, code involves operations not supported by the kernel language, etc.), and our system will simply skip such cases. However, the number of such cases is relatively small as our experiments show.

7. Experiments

In this section we report our experimental results. The goal of the experiments is twofold: first, to quantify the ability of our algorithm to convert Java code into real-world applications and measure the performance of the converted code fragments, and second to explore the limitations of the current implementation.

We have implemented a prototype of QBS. The source code analysis and computation of verification conditions are implemented using the Polyglot compiler framework [25]. We use Sketch as the synthesizer for invariants and postconditions, and Z3 for validating the invariants and postconditions.

7.1 Real-World Evaluation

In the first set of experiments, we evaluated QBS using real-world examples from two large-scale open-source applications, Wilos and itracker, written in Java. Wilos (rev. 1196) [2] is a project management application with 62k LOC, and itracker (ver. 3.0.1) [1] is a software issue management system with 61k LOC. Both applications have multiple contributors with different coding styles, and use the Hibernate ORM library for data persistence operations. We passed in the entire source code of these applications to QBS to identify code fragments. The preprocessor initially found 120 unique code fragments that invoke ORM operations. Of those, it failed to convert 21 of them into the kernel language representation, as they use data structures that are not supported by our prototype (such as Java arrays), or access persistent objects that can escape from multiple control flow points and hence cannot be converted.

Meanwhile, upon manual inspection, we found that those 120 code fragments correspond to 49 distinct code fragments inlined in different contexts. For instance, if A and C both call method B, our system automatically inlines B into the bodies of A and C, and those become two different code fragments. But if all persistent data manipulation happens in B, then we only count one of the two as part of the 49 distinct code fragments. QBS successfully translated 33 out of the 49 distinct code fragments (and those 33 distinct code fragments correspond to 75 original code fragments). The results are summarized in Fig. 13, and the details can be found in Appendix A.

This experiment shows that QBS can infer relational specifications from a large fraction of candidate fragments and convert them

into SQL equivalents. For the candidate fragments that are reported as translatable by QBS, our prototype was able to synthesize postconditions and invariants, and also validate them using the prover. Furthermore, the maximum time that QBS takes to process any one code fragment is under 5 minutes (with an average of 2.1 minutes). In the following, we broadly describe the common types of relational operations that our QBS prototype inferred from the fragments, along with some limitations of the current implementation.

Projections and Selections. A number of identified fragments perform relational projections and selections in imperative code. Typical projections include selecting specific fields from the list of records that are fetched from the database, and selections include filtering a subset of objects using field values from each object (*e.g.*, user ID equals to some numerical constant), and a few use criteria that involve program variables that are passed into the method.

One special case is worth mentioning. In some cases only a single field is projected out and loaded into a set data structure, such as a set of integer values. One way to translate such cases is to generate SQL that fetches the field from all the records (including duplicates) into a list, and eliminate the duplicates and return the set to the user code. Our prototype, however, improves upon that scheme by detecting the type of the result variable and inferring a postcondition involving the unique operator, which is then translated to a SELECT DISTINCT query that avoids fetching duplicate records from the database.

Joins. Another set of code fragments involve join operations. We summarize the join operations in the application code into two categories. The first involves obtaining two lists of objects from two base queries and looping through each pair of objects in a nested for or while loop. The pairs are filtered and (typically) one of the objects from each pair is retained. The running example in Fig. 1 represents such a case. For these cases, QBS translates the code fragment into a relational join of the two base queries with the appropriate join predicate, projection list, and sort operations that preserve the ordering of records in the results.

Another type of join also involves obtaining two lists of objects from two base queries. Instead of a nested loop join, however, the code iterates through each object e from the first list, and searches if e (or one of e's fields) is contained in the second. If true, then e (or some of its fields) is appended to the resulting list. For these cases QBS converts the search operation into a contains expression in the predicate language, after which the expression is translated into a correlated subquery in the form of SELECT * FROM r1, r2 WHERE r1 IN r2, with r1 and r2 being the base queries.

QBS handles both join idioms mentioned above. However, the loop invariants and postconditions involved in such cases tend to be more complex as compared to selections and projections, as illustrated by the running example in Fig. 12. Thus, they require more iterations of synthesis and formal validation before finding a valid solution, with up to 5 minutes in the longest case. Here, the majority of the time is spent in synthesis and bounded verification. We are not aware of any prior techniques that can be used to infer join queries from imperative code, and we believe that more optimizations can be devised to speed up the synthesis process for such cases.

Aggregations. Aggregations are used in fragments in a number of ways. The most straightforward ones are those that return the length of the list that is returned from an ORM query, which are translated into COUNT queries. More sophisticated uses of aggregates include iterating through all records in a list to find the max or min values, or searching if a record exists in a list. Aggregates such as maximum and minimum are interesting as they introduce loop-carried dependencies [5], where the running value of the aggregate is updated conditionally based on the value of the current record as compared to previous ones. By using the top operator from the theory of ordered relations, QBS is able to generate a loop invariant of the form $v = agg(top_i(r))$ and then translate the postcondition into the appropriate SQL query.

As a special case, a number of fragments check for the existence of a particular record in a relation by iterating over all records and setting a result boolean variable to be true if it exists. In such cases, the generated invariants are similar to other aggregate invariants, and our prototype translates such code fragments into SELECT COUNT(*) > 0 FROM ... WHERE e, where e is the expression to check for existence in the relation. We rely on the database query optimizer to further rewrite this query into the more efficient form using EXISTS.

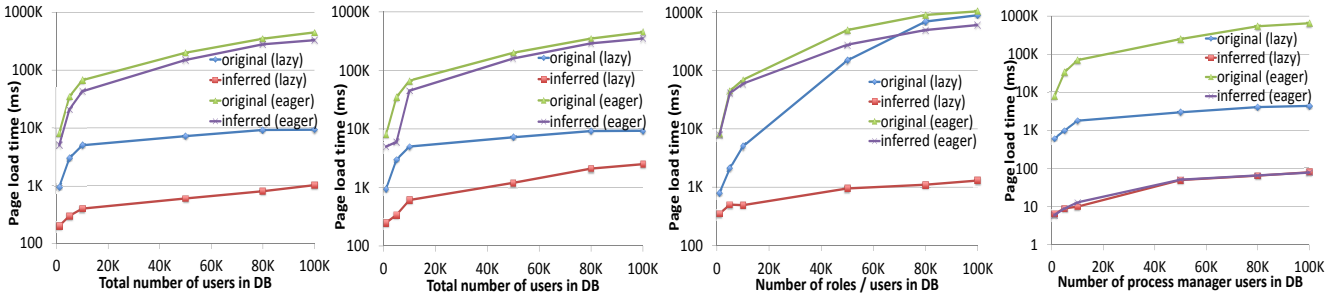
Limitations. We have verified that in all cases where the generated template is expressive enough for the invariants and postconditions, our prototype does indeed find the solution within a preset timeout of 10 minutes. However, there are a few examples from the two applications where our prototype either rejects the input code fragment or fails to find an equivalent SQL expression from the kernel language representation. Fragments are rejected because they involve relational update operations that are not handled by TOR. Another set of fragments include advanced use of types, such as storing polymorphic records in the database, and performing different operations based on the type of records retrieved. Incorporating type information in the theory of ordered relations is an interesting area for future work. There are also a few that QBS fails to translate into SQL, even though we believe that there is an equivalent SQL query without updates. For instance, some fragments involve sorting the input list by Collections.sort, followed by retrieving the last record from the sorted list, which is equivalent to max or min depending on the sort order. Including extra axioms in the theory would allow us to reason about such cases.

7.2 Performance Comparisons

Next, we quantify the amount of performance improvement as a result of query inference. To do so, we took a few representative code fragments for selection, joins, and aggregation, and populated databases with different number of persistent objects. We then compared the performance between the original code and our transformed versions of the code with queries inferred by QBS. Since Hibernate can either retrieve all nested references from the database (eager) when an object is fetched, or only retrieve the top level references (lazy), we measured the execution times for both modes (the original application is configured to use the lazy retrieval mode). The results shown in Fig. 14 compare the time taken to completely load the webpages containing the queries between the original and the QBS inferred versions of the code.

Selection Code Fragment. Fig. 14a and Fig. 14b show the results from running a code fragment that includes persistent data manipulations from fragment #40 in Appendix A. The fragment returns the list of unfinished projects. Fig. 14a shows the results where 10% of the projects stored are unfinished, and Fig. 14b shows the results with 50% unfinished projects. While the original version performs the selection in Java by first retrieving all projects from the database, QBS inferred a selection query in this case. As expected, the query inferred by QBS outperforms the original fragments in all cases as it only needs to retrieve a portion (specifically 10 and 50%) of all persistent objects.

Join Code Fragment. Fig. 14c shows the results from a code fragment with contents from fragment #46 in Appendix A (which is the same as the example from Fig. 1). The fragment returns the projection of User objects after a join of Roles and Users in the



(a) Selection with 10% selectivity (b) Selection with 50% selectivity (c) Join code fragment (d) Aggregation code fragment

Figure 14: Webpage load times comparison of representative code fragments

database on the roleId field. The original version performs the join by retrieving all User and Role objects and joining them in a nested loop fashion as discussed in Sec. 2. The query inferred by QBS however, pushes the join and projection into the database. To isolate the effect of performance improvement due to query selectivity (as in the selection code fragment), we purposefully constructed the dataset so that the query returns all User objects in the database in all cases, and the results show that the query inferred by QBS version still has much better performance than the original query. This is due to two reasons. First, even though the number of User objects returned in both versions are the same, the QBS version does not need to retrieve any Role objects since the projection is pushed into the database, unlike the the original version. Secondly, thanks to the automatically created indices on the Role and User tables by Hibernate, the QBS version essentially transforms the join implementation from a nested loop join into a hash join, *i.e.*, from an O(n^2) to an O(n) implementation, thus improving performance asymptotically.

Aggregation Code Fragment. Finally, Fig. 14d shows the results from running code with contents from fragment #38, which returns the number of users who are process managers. In this case, the original version performs the counting by bringing in all users who are process managers from the database, and then returning the size of the resulting list. QBS, however, inferred a COUNT query on the selection results. This results in multiple orders of magnitude performance improvement, since the QBS version does not need to retrieve any objects from the database beyond the resulting count.

7.3 Advanced Idioms

In the final set of experiments, we used synthetic code fragments to demonstrate the ability of our prototype to translate more complex expressions into SQL. Although we did not find such examples in either of our two real-world applications, we believe that these can occur in real applications.

Hash Joins. Beyond the join operations that we encountered in the applications, we wrote two synthetic test cases for joins that join relations r and s using the predicate r.a = s.b, where a and b are integer fields. In the first case, the join is done via hashing, where we first iterate through records in r and build a hashtable, whose keys are the values of the a field, and where each key maps to a list of records from r that has that corresponding value of a. We then loop through each record in s to find the relevant records from r to join with, using the b field as the look up key. QBS currently models hashtables using lists, and with that our prototype is able recognize this process as a join operation and convert the fragment accordingly, similar to the join code fragments mentioned above.

Sort-Merge Joins. Our second synthetic test case joins two lists by first sorting r and s on fields a and b respectively, and then iterating through both lists simultaneously. We advance the scan of r as long as the current record from r is less than (in terms of fields a and b) the current record from s, and similarly advance the scan of s as long as the current s record is less than the current r record. Records that represent the join results are created when the current record from r equals to that from s on the respective fields. Unfortunately, our current prototype fails to translate the code fragment into SQL, as the invariants for the loop cannot be expressed using the current the predicate language, since that involves expressing the relationship between the current record from r and s with all the records that have been previously processed.

Iterating over Sorted Relations. We next tested our prototype with two usages of sorted lists. We created a relation with one unsigned integer field id as primary key, and sorted the list using the sort method from Java. We subsequently scanned through the sorted list as follows:

```
List records = Query("SELECT id FROM t");
List results = new ArrayList();
Collections.sort(records); // sort by id
for (int i = 0; i < 10; ++i)
{ results.add(records.get(i)); }
```

Our prototype correctly processes this code fragment by translating it into SELECT id FROM t ORDER BY id LIMIT 10. However, if the loop is instead written as follows:

```
List records = Query("SELECT id FROM t");
List results = new ArrayList();
Collections.sort(records); // sort by id
int i = 0;
while (records.get(i).id < 10)
{ results.add(records.get(i)); ++i;}
```

The two loops are equivalent since the id field is a primary key of the relation, and thus there can at most be 10 records retrieved. However, our prototype is not able to reason about the second code fragment, as that requires an understanding of the schema of the relation, and that iterating over id in this case is equivalent to iterating over i in the first code fragment. Both of which require additional axioms to be added to the theory before such cases can be converted.

8. Related Work

Inferring relational specifications from imperative code was studied in [37, 38]. The idea is to compute the set of data access paths that the imperative code traverses using abstract interpretation, and replace the imperative code with SQL queries. The analysis is applicable to recursive function calls, but does not handle code with loop-carried dependencies, or those with join or aggregation. It is unclear how modeling relational operations as access paths can be extended to handle such cases. Our implementation is able to infer both join and aggregation from imperative code. We currently do

not handle recursive function calls, although we have not encountered such uses in the applications used in our experiments.

Modeling relational operations. Our ability to infer relational specifications from imperative code relies on using the TOR to connect the imperative and relational worlds. There are many prior work in modeling relational operations, for instance using bags [9], sets [22], and nested relational calculus [39]. There are also theoretical models that extend standard relational algebra with order, such as [4, 24]. Our work does not aim to provide a new theoretical model. Instead, one key insight of our work is that TOR is the right abstraction for the query inference, as they are similar to the interfaces provided by the ORM libraries in the imperative code, and allow us to design a sound and precise transformation into SQL.

To our knowledge, our work is the first to address the ordering of records in relational transformations. Record ordering would not be an issue if the source program only operated on orderless data structures such as sets or did not perform any joins. Unfortunately, most ORM libraries provide interfaces based on ordered data structures, and imperative implementations of join proves to be common at least in the benchmarks we studied.

Finding invariants to validate transformations. Our verification-based approach to finding a translatable postcondition is similar to earlier work [20, 21], although they acknowledge that finding invariants is difficult. Similar work has been done for general-purpose language compilers [33]. Scanning the source code to generate the synthesis template is inspired by the PINS algorithm [32], although QBS does not need user intervention. There has been earlier work on automatically inferring loop invariants, such as using predicate refinement [14] or dynamic detection [13, 18].

Constraint-based synthesis. Constraint-based program synthesis has been an active topic of research in recent years. For instance, there has been work on using synthesis to discover invariants [31]. Our work differs from previous approaches in that we only need to find invariants and postconditions that are *strong enough* to validate the transformation, and our predicate language greatly prunes the space of invariants to those needed for common relational operations. Synthesis has also been applied in other domains, such as generating data structures [29], processor instructions [15], and learning queries from examples [8].

Integrated query languages. Integrating application and database query languages into has been an active research area, with projects such as LINQ [23], Kleisli [39], Links [10], JReq [20], the functional language proposed in [9], Ferry [17], and DBPL [28]. These solutions embed database queries in imperative programs without ORM libraries. Unfortunately, many of them do not support all relational operations, and the syntax of such languages resemble SQL, thus developers still need to learn new programming paradigms and rewrite existing applications.

Improving performance of database programs by code transformation. There is also work in improving application performance by transforming loops to allow query batching [26], and pushing computations into the database [7]. Our work is orthogonal to this line of research. After converting portions of the source code into SQL queries, such code transformation can be applied to gain additional performance improvement.

9. Conclusions

In this paper, we presented QBS, a system for inferring relational specifications from imperative code that retrieves data using ORM libraries. Our system automatically infers loop invariants and postconditions associated with the source program, and converts the validated postcondition into SQL queries. Our approach is both sound and precise in preserving the ordering of records. We developed a theory of ordered relations that allows efficient encoding of relational operations into a predicate language, and we demonstrated the applicability using a set of real-world code examples.

References

[1] itracker Issue Management System. `http://itracker.sourceforge.net/index.html`.

[2] Wilos Orchestration Software. `http://www.ohloh.net/p/6390`.

[3] z3 Theorem Prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3`.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proc. CC*, pages 233–246, 1984.

[6] A. Blass and Y. Gurevich. Inadequacy of computable loop invariants. *ACM Trans. Comput. Log.*, 2(1):1–11, 2001.

[7] A. Cheung, O. Arden, S. Madden, and A. C. Myers. Automatic partitioning of database applications. *PVLDB*, 5, 2011.

[8] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. In *Proc. CIKM*, pages 1732–1736, 2012.

[9] E. Cooper. The script-writer's dream: How to write great sql in your own language, and be sure it will succeed. In *Proc. DBPL Workshop*, pages 36–51, 2009.

[10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. Int.l Symp. on Formal Methods for Components and Objects*, pages 266–296, 2006.

[11] D. Déharbe, P. Fontaine, S. Merz, and B. W. Paleo. Exploiting symmetry in smt problems. In *Proc. Int.l Conf. on Automated Deduction*, pages 222–236, 2011.

[12] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. ICSE*, pages 213–224, 1999.

[14] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL*, pages 191–202, 2002.

[15] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proc. PLDI*, pages 441–452, 2012.

[16] D. Gries. *The Science of Programming*. Springer, 1987.

[17] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *Proc. SIGMOD*, pages 1063–1066, 2009.

[18] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Proc. CAV*, pages 634–640, 2009.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[20] M.-Y. Iu, E. Cecchet, and W. Zwaenepoel. Jreq: Database queries in imperative languages. In *Proc. CC*, pages 84–103, 2010.

[21] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: a mapreduce query optimizer. In *Proc. EuroSys*, pages 251–264, 2010.

[22] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[23] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. SIGMOD*, pages 706–706, 2006.

[24] W. Ng. An extension of the relational data model to incorporate ordered domains. *Trans. Database Syst.*, 26(3):344–383, Sept. 2001.

[25] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. CC*, pages 138–152, 2003.

[26] K. Ramachandra, R. Guravannavar, and S. Sudarshan. Program analysis and transformation for holistic optimization of database applications. In *Proc. SOAP Workshop*, pages 39–44, 2012.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. POPL*, pages 105–118, 1999.

[28] J. W. Schmidt and F. Matthes. The DBPL project: Advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.

[29] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proc. FSE*, pages 289–299, 2011.

[30] A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proc. ASPLOS*, pages 404–415, 2006.
[31] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *Proc. PLDI*, pages 223–234, 2009.
[32] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *Proc. PLDI*, pages 492–503, 2011.
[33] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proc. PLDI*, pages 111–121, 2010.
[34] E. Torlak and D. Jackson. Kodkod: a relational model finder. In *Proc. TACAS*, pages 632–647, 2007.
[35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *Proc. PLDI*, pages 87–97, 2009.
[36] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proc. OOPSLA*, pages 187–206, 1999.
[37] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *Proc. POPL*, pages 199–210, 2007.
[38] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *Proc. OOPSLA*, pages 19–36, 2008.
[39] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.
[40] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. POPL*, pages 351–363, 2005.

A. Persistent Data Code Fragment Details

In this section we describe the code fragments from real-world examples that are used in our experiments. The table below shows the details of the 49 distinct code fragments as mentioned in Sec. 7.1. The times reported correspond to the time required to synthesize the invariants and postconditions. The time taken for the other initial analysis and SQL translation steps are negligible.

wilos code fragments

#	Java Class Name	Line	Oper.	Status	Time (s)
17	ActivityService	401	A	†	–
18	ActivityService	328	A	†	–
19	AffectedtoDao	13	B	✓	72
20	ConcreteActivityDao	139	C	*	–
21	ConcreteActivityService	133	D	†	–
22	ConcreteRoleAffectationService	55	E	✓	310
23	ConcreteRoleDescriptorService	181	F	✓	290
24	ConcreteWorkBreakdownElementService	55	G	†	–
25	ConcreteWorkProductDescriptorService	236	F	✓	284
26	GuidanceService	140	A	†	–
27	GuidanceService	154	A	†	–
28	IterationService	103	A	†	–
29	LoginService	103	H	✓	125
30	LoginService	83	H	✓	164
31	ParticipantBean	1079	B	✓	31
32	ParticipantBean	681	H	✓	121
33	ParticipantService	146	E	✓	281
34	ParticipantService	119	E	✓	301
35	ParticipantService	266	F	✓	260
36	PhaseService	98	A	†	–
37	ProcessBean	248	H	✓	82
38	ProcessManagerBean	243	B	✓	50
39	ProjectService	266	K	*	–
40	ProjectService	297	A	✓	19
41	ProjectService	338	G	†	–
42	ProjectService	394	A	✓	21
43	ProjectService	410	A	✓	39
44	ProjectService	248	H	✓	150
45	RoleDao	15	I	*	–
46	RoleService	15	E	✓	150
47	WilosUserBean	717	B	✓	23
48	WorkProductsExpTableBean	990	B	✓	52
49	WorkProductsExpTableBean	974	J	✓	50

itracker code fragments

#	Java Class Name	Line	Operation	Status	Time (s)
1	EditProjectFormActionUtil	219	F	✓	289
2	IssueServiceImpl	1437	D	✓	30
3	IssueServiceImpl	1456	L	*	–
4	IssueServiceImpl	1567	C	*	–
5	IssueServiceImpl	1583	M	✓	130
6	IssueServiceImpl	1592	M	✓	133
7	IssueServiceImpl	1601	M	✓	128
8	IssueServiceImpl	1422	D	✓	34
9	ListProjectsAction	77	N	*	–
10	MoveIssueFormAction	144	K	*	–
11	NotificationServiceImpl	568	O	✓	57
12	NotificationServiceImpl	848	A	✓	132
13	NotificationServiceImpl	941	H	✓	160
14	NotificationServiceImpl	244	O	✓	72
15	UserServiceImpl	155	M	✓	146
16	UserServiceImpl	412	A	✓	142

where:
A: selection of records
B: return literal based on result size
C: retrieve the max / min record by first sorting and then returning the last element
D: projection / selection of records and return results as a set
E: nested-loop join followed by projection
F: join using contains
G: type-based record selection
H: check for record existence in list
I: record selection and only return the one of the records if multiple ones fulfill the selection criteria
J: record selection followed by count
K: sort records using a custom comparator
L: projection of records and return results as an array
M: return result set size
N: record selection and in-place removal of records
O: retrieve the max / min record

✓ indicates those that are translated by. QBS
* indicates those that QBS failed to find invariants for.
† indicates those that are rejected by QBS due to TOR / preprocessing limitations.

B. TOR Expression Equivalences and Definition of Trans

Before defining Trans, we first list the set of TOR expression equivalences that are used in the definition.

Theorem 2 (Operator Equivalence). *The following equivalences hold, both in terms of the contents of the relations and also the ordering of the records in the relations:*

- $\sigma_{\varphi}(\pi_{\ell}(r)) = \pi_{\ell}(\sigma_{\varphi}(r))$
- $\sigma_{\varphi_2}(\sigma_{\varphi_1}(r)) = \sigma_{\varphi'}(r)$, *where* $\varphi' = \varphi_2 \wedge \varphi_1$
- $\pi_{\ell_2}(\pi_{\ell_1}(r)) = \pi_{\ell'}(r)$, *where* ℓ' *is the concatenation of all the fields in* ℓ_1 *and* ℓ_2.
- $\mathsf{top}_e(\pi_{\ell}(r)) = \pi_{\ell}(\mathsf{top}_e(r))$
- $\mathsf{top}_{e_2}(\mathsf{top}_{e_1}(r)) = \mathsf{top}_{\mathsf{max}(e_1,e_2)}(r)$
- $\bowtie_{\varphi}(r_1, r_2) = \sigma_{\varphi'}(\bowtie_{\mathsf{True}}(r_1, r_2))$, *i.e., joins can be converted into cross products with selections with proper renaming of fields.*
- $\bowtie_{\varphi}(\mathsf{sort}_{\ell_1}(r_1), \mathsf{sort}_{\ell_2}(r_2)) = \mathsf{sort}_{\ell_1:\ell_2}(\bowtie_{\varphi}(r_1, r_2))$
- $\bowtie_{\varphi}(\pi_{\ell_1}(r_1), \pi_{\ell_2}(r_2)) = \pi_{\ell'}(\bowtie_{\varphi}(r_1, r_2))$, *where* ℓ' *is the concatenation of all the fields in* ℓ_1 *and* ℓ_2.

Except for the equivalences involving sort, the other ones can be proven easily from the axiomatic definitions.

B.1 Definition of Trans

Let $s = \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b)))$. Trans is defined on expressions whose subexpressions (if any) are in translatable form, so we have to consider cases where the sub-expressions are either s or $\mathsf{top}_e(s)$. Each case is defined below.

$$\boxed{\mathsf{Query}(\ldots)}$$

$$\mathsf{Trans}(\mathsf{Query}(\ldots))) = \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_{\mathsf{True}}(\mathsf{Query}(\ldots))))$$

where ℓ projects all the fields from the input relation.

$$\boxed{\pi_{\ell_2}(t)}$$

$$\begin{aligned}\mathsf{Trans}(\pi_{\ell_2}(s)) &= \pi_{\ell'}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b)))\\ \mathsf{Trans}(\pi_{\ell_2}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell'}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))))\end{aligned}$$

where ℓ' is the composition of ℓ_π and ℓ_2.

$$\boxed{\sigma_{\varphi_2}(t)}$$

$$\begin{aligned}\mathsf{Trans}(\sigma_{\varphi_2}(s)) &= \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi\wedge\varphi_2}(b)))\\ \mathsf{Trans}(\sigma_{\varphi_2}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi\wedge\varphi_2}(b))))\end{aligned}$$

$$\boxed{\bowtie_{\varphi_\bowtie}(t_1, t_2))}$$

$$\mathsf{Trans}(\bowtie_{\varphi_\bowtie}(s_1, s_2)) = \pi_{\ell'_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_{\varphi'_\sigma}(\bowtie_{\mathsf{True}}(b_1, b_2))))$$

where $\varphi'_\sigma = \varphi_{\sigma_1} \wedge \varphi_{\sigma_2} \wedge \varphi_\bowtie$ with field names properly renamed, $\ell'_s = \mathsf{cat}(\ell_{s_1}, \ell_{s_2})$, and $\ell'_\pi = \mathsf{cat}(\ell_{\pi_1}, \ell_{\pi_2})$.

$$\begin{aligned}&\mathsf{Trans}(\bowtie_\varphi(\mathsf{top}_e(s_1), \mathsf{top}_e(s_2)))\\ &\quad = \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_\varphi(\bowtie_{\mathsf{True}}(\mathsf{top}_e(s_1), \mathsf{top}_e(s_2))))))\end{aligned}$$

where ℓ contains all the fields from s_1 and s_2.

$$\boxed{\mathsf{top}_{e_2}(t)}$$

$$\begin{aligned}\mathsf{Trans}(\mathsf{top}_{e_2}(s)) &= \mathsf{top}_{e_2}(s)\\ \mathsf{Trans}(\mathsf{top}_{e_2}(\mathsf{top}_{e_1}(s))) &= \mathsf{top}_{e'}(s)\end{aligned}$$

where e' is the minimum value of e_1 and e_2.

$$\boxed{\mathsf{agg}(t)}$$

$$\begin{aligned}\mathsf{Trans}(\mathsf{agg}(s)) &= \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_{\mathsf{True}}(\mathsf{agg}(s))))\\ \mathsf{Trans}(\mathsf{agg}(\mathsf{top}_e(s))) &= \pi_\ell(\mathsf{sort}_{[\,]}(\sigma_{\mathsf{True}}(\mathsf{agg}(s))))\end{aligned}$$

where ℓ contains all the fields from s.

$$\boxed{\mathsf{sort}_{\ell_{s_2}}(t)}$$

$$\begin{aligned}\mathsf{Trans}(\mathsf{sort}_{\ell_{s_2}}(s)) &= \pi_{\ell_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_\varphi(b)))\\ \mathsf{Trans}(\mathsf{sort}_{\ell_{s_2}}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_\varphi(b))))\end{aligned}$$

where $\ell'_s = \mathsf{cat}(\ell_s, \ell_{s_2})$.

C. TOR Axioms

Below is a listing of all the axioms currently defined in the theory of ordered relations (TOR). $\mathsf{cat}(\ell_1, \ell_2)$ concatenates the contents of lists ℓ_1 and ℓ_2.

$$\boxed{\mathsf{size}}$$

$$\frac{r = [\,]}{\mathsf{size}(r) = 0} \qquad \frac{r = h : t}{\mathsf{size}(r) = 1 + \mathsf{size}(t)}$$

$$\boxed{\mathsf{get}}$$

$$\frac{i = 0 \quad r = h : t}{\mathsf{get}_i(r) = h} \qquad \frac{i > 0 \quad r = h : t}{\mathsf{get}_i(r) = \mathsf{get}_{i-1}(t)}$$

$$\boxed{\mathsf{append}}$$

$$\frac{r = [\,]}{\mathsf{append}(r, t) = [t]} \qquad \frac{r = h : t}{\mathsf{append}(r, t') = h : \mathsf{append}(t, t')}$$

$$\boxed{\mathsf{top}}$$

$$\frac{r = [\,]}{\mathsf{top}_r(i) = [\,]} \qquad \frac{i = 0}{\mathsf{top}_r(i) = [\,]} \qquad \frac{i > 0 \quad r = h : t}{\mathsf{top}_r(i) = h : \mathsf{top}_t(i-1)}$$

$$\boxed{\mathsf{join}\ (\bowtie)}$$

$$\frac{r_1 = [\,]}{\bowtie_\varphi(r_1, r_2) = [\,]} \qquad \frac{r_2 = [\,]}{\bowtie_\varphi(r_1, r_2) = [\,]}$$

$$\frac{r_1 = h : t}{\bowtie_\varphi(r_1, r_2) = \mathsf{cat}(\bowtie'_\varphi(h, r_2),\ \bowtie_\varphi(t, r_2))}$$

$$\frac{r_2 = h : t \quad \varphi(e, h) = \mathsf{True}}{\bowtie'_\varphi(e, r_2) = (e, h) : \bowtie'_\varphi(e, t)} \qquad \frac{r_2 = h : t \quad \varphi(e, h) = \mathsf{False}}{\bowtie'_\varphi(e, r_2) = \bowtie'_\varphi(e, t)}$$

$$\boxed{\mathsf{projection}\ (\pi)}$$

$$\frac{r = [\,]}{\pi_\ell(r) = [\,]} \qquad \frac{r = h : t \quad f_i \in \ell \quad h.f_i = e_i}{\pi_\ell(r) = \{f_i = e_i\} : \pi_\ell(t)}$$

$$\boxed{\mathsf{selection}\ (\sigma)}$$

$$\frac{r = [\,]}{\sigma_\varphi(r) = [\,]} \qquad \frac{r = h : t \quad \varphi(h) = \mathsf{True}}{\sigma_\varphi(r) = h : \sigma_\varphi(t)} \qquad \frac{r = h : t \quad \varphi(h) = \mathsf{False}}{\sigma_\varphi(r) = \sigma_\varphi(t)}$$

$$\boxed{\mathsf{sum}}$$

$$\frac{r = [\,]}{\mathsf{sum}(r) = 0} \qquad \frac{r = h : t}{\mathsf{sum}(r) = h + \mathsf{sum}(t)}$$

$$\boxed{\mathsf{max}}$$

$$\frac{r = [\,]}{\mathsf{max}(r) = -\infty} \qquad \frac{r = h : t \quad h > \mathsf{max}(t)}{\mathsf{max}(r) = h} \qquad \frac{r = h : t \quad h \leq \mathsf{max}(t)}{\mathsf{max}(r) = \mathsf{max}(t)}$$

$$\boxed{\mathsf{min}}$$

$$\frac{r = [\,]}{\mathsf{min}(r) = \infty} \qquad \frac{r = h : t \quad h < \mathsf{min}(t)}{\mathsf{min}(r) = h} \qquad \frac{r = h : t \quad h \geq \mathsf{min}(t)}{\mathsf{min}(r) = \mathsf{min}(t)}$$

$$\boxed{\mathsf{contains}}$$

$$\frac{r = [\,]}{\mathsf{contains}(e, r) = \mathsf{False}}$$

$$\frac{e = h \quad r = h : t}{\mathsf{contains}(e, r) = \mathsf{True}} \qquad \frac{e \neq h \quad r = h : t}{\mathsf{contains}(e, r) = \mathsf{contains}(e, t)}$$