# Extracting Queries by Static Analysis of Transparent Persistence*

Ben Wiedermann and William R. Cook

Department of Computer Sciences, The University of Texas at Austin
{ben,wcook}@cs.utexas.edu

## Abstract

Transparent persistence promises to integrate programming languages and databases by allowing procedural programs to access persistent data with the same ease as non-persistent data. When the data is stored in a relational database, however, transparent persistence does not naturally leverage the performance benefits of relational query optimization. We present a program analysis that combines the benefits of both approaches by extracting database queries from programs with transparent access to persistent data. The analysis uses a sound abstract interpretation of the original program to approximate the data traversal paths in the program and the conditions under which the paths are used. The resulting paths are then converted into a query, and the program is simplified by removing redundant tests. We study an imperative kernel language with read-only access to persistent data and identify the conditions under which the transformations can be applied. This analysis approach promises to combine the software engineering benefits of transparent data persistence with the performance benefits of database query optimization.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors; H.2.3 [*Database Management*]: Languages

***General Terms*** Languages, Performance

***Keywords*** Programming Languages, Databases, Static Analysis

## 1. Introduction

The effective integration of programming languages and databases is a long-standing and critical open problem. From a programming language viewpoint, databases manage *persistent* data, which has a lifetime longer than the execution of an individual program. Ideally a unified programming model should be applicable to both persistent and non-persistent data. This goal has been pursued for the last 30 years in numerous forms, including orthogonal persistence [2, 3, 4, 25, 30], object-relational mapping [16, 22, 28, 34], and object-oriented databases [12, 15, 27]. Despite differences in particular details, these approaches all share the goal of *transparent persistence*—a programming paradigm wherein the programmer need not distinguish between persistent and non-persistent values.

Transparent persistence can be added to most any language by extending the concepts of automatic memory management and garbage collection to the management of persistent data: by identifying a persistent root object, any object or value reachable from the root is also persistent [4]. For example, the Java program in Fig. 1 manipulates a collection of employee objects associated with a root object. If root identifies a persistent store of objects, then the employee objects may be loaded from that store. However, the program's result is independent of whether root is persistent or not.

```
for (Employee e : root.employees) {
  if (e.salary > 65000) {
    print(e.name + ": " + e.manager.name);
}}
```

**Figure 1.** A program using transparent persistence.

```
// define an explicit query
String query = "from Employee e
  left join fetch e.manager
  where e.salary > 65000";
// execute the query
List result = executeQuery(query);
for (Employee e : result.list()) {
    // no test required: all elements already satisfy
    // the condition salary > 65000
    print(e.name + ": " + e.manager.name);
}
```

**Figure 2.** Explicit query execution using Hibernate.

This kind of transparent persistence does not easily leverage the power of database query optimization. Database optimizations work best when records are loaded in bulk and conditions for selecting records are executed in the database rather than the procedural program. The mismatch between one-at-a-time processing in procedural language and bulk data processing in query operations is called "impedance mismatch" [26]. To solve this problem, many persistence models allow programmers to execute explicit queries. For example, Fig. 2 uses Hibernate, an object-relational mapping tool, and its query language HQL [22] to execute an explicit query. The executeQuery method is a (unspecified) helper function that hides some of the details of executing a query with Hibernate. The query returns only employees with salary greater than $65,000; the prefetch clause **left join fetch** e.manager indicates that each employee's manager should also be loaded. The **if** statement in Fig. 1 is not needed in Fig. 2 because the query's **where** clause ensures the query only returns employees for which the test is true.

Although the programs in Fig. 1 and Fig. 2 print the same results, they have different performance and software engineering benefits. In the transparent persistence version, all employees will be loaded even though only those with salary greater than $65,000 are printed. Manager objects will be loaded individually, because the persistence layer cannot predict which ones will be needed.

* This work was supported by the National Science Foundation under Grant No. 0448128.



*POPL'07* January 17–19, 2007, Nice, France.

In the Hibernate version, the underlying relational query optimizer will likely use an index to locate all employees whose salary is greater than $65,000. The optimized version runs in time proportional to the size of the query result, rather than the total number of employees and may be orders of magnitude faster [9].

Despite its performance benefits, there are some drawbacks to the Hibernate version. Query strings are not checked at compile time for syntax or type safety, and they reduce modularity and increase the complexity of programming. Proposals to address these problems [7, 11, 20] either reduce or do not address the transparency of persistence. There is also a subtle dependency between the query and the code: the prefetch clause is logically redundant with the use of the employee's manager in the **print** method.

This paper describes a static analysis technique that allows a programming language with transparent persistence to leverage the power of query optimization. Our approach automatically partitions programs by extracting data traversals and conditions into a query, and removing them from the program—essentially transforming the program in Fig. 1 into the program in Fig. 2. This transformation requires that the extracted query return a structural subset of the original database. The analysis creates queries written in a subset of the Object Query Language (OQL) [8] that can express these kinds of queries.

Our analysis consists of three parts. The first part (Section 3) identifies the traversals used in the program; these traversals specify the data that must be loaded by the query. The second part (Section 4) identifies the conditions under which data is used, so that the conditions can be included in the query. In the final part (Section 5) the individual conditions on the use of fields are promoted to apply to entire records, and a query is created. This final step also modifies the program to use the results of the query, and eliminates redundant **if** statements.

The primary contribution of this paper is a new approach to optimization of transparent persistence by extracting queries from imperative programs. This result is based on a sound abstract interpretation of programs, together with techniques for converting the resulting abstract values into queries and simplifying the original program. We have developed a prototype implementation of the analysis and applied it to simple examples to demonstrate its viability. While this work re-opens an important line of research, there are many topics left to future work. In particular, we have not analyzed the performance of the analysis or the transformed programs—although the performance gains from query optimization are well-known. We have not applied the analysis to large programs with procedures, or addressed the problem of identifying where in a large program the analysis should be applied. Complex query behaviors, like aggregation, exists queries, and database mutations (creations, updates, and deletions) are not considered. We expect that the current work will serve as a solid foundation for ongoing work on these problems, with the goal of combining the software engineering benefits of transparent persistence and the performance benefits of query optimization.

## 2. A Kernel Language with Persistent Data

We study a simple imperative language with records and access to persistent data. The persistent data is an instance of an Entity-Relationship (ER) Model [10], which provides natural mappings to both relational databases [5] and class models in UML/object-oriented programming [37]. A persistent value is a record, or labeled product, whose fields are either basic values or references to other records (these are called "attributes" and "relationships" in an ER model). A reference/relationship field may be either single-valued or multi-valued. Multi-valued relationships correspond to collection objects in object-oriented programming. The language expresses key concepts in practical orthogonally persistent object-oriented languages, but it also has several restrictions. Only the structural representation of data is considered, not behavioral methods, and the language contains no procedures. We do not model the three-valued logic of null values that is used in SQL, but assume that a value is defined for every persistent element a program accesses. While the language supports imperative update of local variables, persistent data is read-only. We believe these restrictions to be reasonable, as the current work is designed to introduce a technique for extracting procedural queries. Section 7 discusses extensions to support interprocedural analysis, analysis of more complicated query idioms, and creation, update, or deletion of persistent data.

$$
\begin{array}{rcl}
l \in \mathit{Variable} & & \\
f \in \mathit{Field} & & \\
e \in \mathit{Expression} & ::= & l \mid e.f \mid \mathsf{op}_n(e_1, \ldots, e_n) \\
\mathsf{op}_0 \in \mathit{Constant} & ::= & \textbf{true} \mid \textbf{false} \mid \mathit{number} \mid \mathit{string} \\
\mathsf{op}_1 & ::= & \neg \mid \textbf{print} \\
\mathsf{op}_2 & ::= & \wedge \mid \vee \mid > \mid < \mid = \mid \geq \mid \leq \mid \neq \\
c \in \mathit{Command} & ::= & \textbf{skip} \mid l := e \mid c; c \\
& \mid & \textbf{if } e \textbf{ then } c \; [\textbf{else } c] \\
& \mid & \textbf{for } l \textbf{ in } e \textbf{ do } c
\end{array}
$$

**Figure 3.** Syntax of a persistent data kernel language.

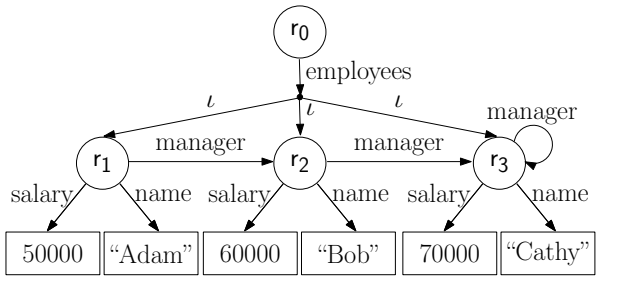


**Figure 4.** An object graph example.

### 2.1 Syntax

The abstract syntax of the kernel language is defined in Fig. 3. The traversal expression $e.f$ projects a field $f$ of a record $e$. The value of $e.f$ can be a simple value, or references to one or more records.

Persistent data is introduced through a special `root` variable that refers to a record representing persistent data [4]. Any value that is reachable from the root is also persistent. As mentioned above, no constructs create or modify persistent records; all records are loaded from the persistent store.

Primitive functions $\mathsf{op}_n$ have a specified number of arguments $n$. Infix notation is used where appropriate.

The **for** command allows iteration over the elements of a collection. For simplicity, iteration is supported only for multi-valued relationship fields—that is, collections of persistent objects. The language could easily be extended to allow iteration of non-persistent collections or basic values.

A simple static type system for records is assumed for this language [32]; programs are assumed to be well typed.

### 2.2 Values

A program operates over the domain:

$$v \in \mathit{Value} = \mathit{Basic} + \mathit{RecordID} + \mathit{RecordID}^*$$

$$\langle l, \sigma \rangle \to \sigma[l] \qquad \text{(S-VAR)}$$

$$\langle \textbf{skip}, \sigma \rangle \to \sigma \qquad \text{(S-SKIP)}$$

$$\frac{\langle e, \sigma \rangle \to r}{\langle e.f, \sigma \rangle \to \mathit{Load}(r, f)} \qquad \text{(S-TRAVERSE)}$$

$$\frac{\langle e_i, \sigma \rangle \to v_i \quad \textbf{for } i \in \{1, \ldots, n\}}{\langle \mathsf{op}_n(e_1, \ldots, e_n), \sigma \rangle \to f_{\mathsf{op}_n}(v_1, \ldots, v_n)} \qquad \text{(S-OP)}$$

$$\frac{\langle c_1, \sigma \rangle \to \sigma' \qquad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1; c_2, \sigma \rangle \to \sigma''} \qquad \text{(S-SEQ)}$$

$$\frac{\langle e, \sigma \rangle \to v}{\langle l{:=}e, \sigma \rangle \to [l \mapsto v]\sigma} \qquad \text{(S-ASSIGN)}$$

$$\frac{\langle e, \sigma \rangle \to \mathit{true} \qquad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \sigma'} \qquad \text{(S-IFT)}$$

$$\frac{\langle e, \sigma \rangle \to \mathit{false} \qquad \langle c_2, \sigma \rangle \to \sigma'}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \sigma'} \qquad \text{(S-IFF)}$$

$$\frac{\langle e, \sigma \rangle \to \{r_1, \ldots, r_n\} \qquad \langle c, [l \mapsto r_i]\sigma_i \rangle \to \sigma_{i+1} \quad \textbf{for } i \in \{1, \ldots, n\}}{\langle \textbf{for } l \textbf{ in } e \textbf{ do } c, \sigma_1 \rangle \to \sigma_{n+1} \backslash l} \qquad \text{(S-FOR)}$$

**Figure 5.** Operational semantics of the kernel language.

where $\mathit{Basic}$ is the domain of basic values (integers, Booleans, and strings), and $\mathit{RecordID}$ is the domain of *record identifiers* that reference persistent database values. When a program traverses a record identifier, a runtime function $\mathit{Load} :: \mathit{RecordID} \times \mathit{Field} \to \mathit{Value}$ retrieves the corresponding record's field value(s). A special record identifier $\mathsf{r}_0$ corresponds to the persistent store's root, and the store's structure is the graph formed by the transitive closure of traversals from $\mathsf{r}_0$.

Figure 4 illustrates a persistent object graph, against which the program in Fig. 1 can be evaluated. The graph's solid dot denotes a collection of record identifiers, where the target of each outgoing edge is a member of the collection. Each of these edges is implicitly labeled with an *iterator field name* $\iota$, which identifies distinct elements of the collection.

### 2.3 Semantics

Figure 5 defines a big-step operational semantics for the kernel language. A store $\sigma$ maps variables to values. The evaluation relations for expressions $\langle e, \sigma \rangle \to v$ and commands $\langle c, \sigma \rangle \to \sigma'$ follow standard form. All programs begin computation with a store $\sigma_0$ that maps the variable root to the persistent value $\mathsf{r}_0$.

Rule S-VAR retrieves a variable's value from the store. For operations, rule S-OP evaluates the operands, then applies the operator's function to the result. The functions $f_{\textbf{true}}, f_{\neg}, f_{<}$, etc., have the standard mathematical meanings. Note that these functions are defined so that they return only primitive values, not record identifiers. Rules S-SKIP, S-ASSIGN, S-SEQ, S-IFT and S-IFF are standard. The expression $[l \mapsto v]\sigma$ denotes $\sigma$ updated so that $\sigma[l] = v$.

Rule S-TRAVERSE loads persistent data. If expression $e$ evaluates to the record identifier $r$, then the expression $e.f$ evaluates to the result of calling $\mathit{Load}(r, f)$.

Rule S-FOR defines iteration over a collection of record identifiers. For each record identifier $r_i$ in the collection, the **for** command's body $c$ is evaluated in a new store $\sigma_i$ that maps the loop variable $l$ to record identifier $r_i$. The result of the entire command is the final store produced $\sigma_{n+1}$. A loop variable is defined only in the loop's body, so the variable is removed from the final store.

One subtle difference between the semantics of object-oriented programming languages and relational databases is that programming languages assume that collections have an inherent order, whereas databases do not. For our purposes, we assume a default order exists for every database collection and that programs iterate over collections in that order.

Output from the **print** function is modeled by a special variable **output**; the **print** function simply concatenates onto the end of this variable.

Evaluating the example program in Fig. 1 against the persistent data in Fig. 4 generates a final store with the following mappings:

$$\mathsf{root} \mapsto r_0, \ \textbf{output} \mapsto \text{“Cathy : Cathy”}$$

### 2.4 Operational Semantics with Explicit Used-Set

Our analysis summarizes the set of persistent values a program uses. These values—which we refer to as the program's *used-set*—can then be loaded in bulk before the program needs them. The operational semantics of the base language is extended in Fig. 6 to keep track of a computation's used-set. The modified semantics has evaluation relations $\langle e, \sigma \rangle \to \langle v, \rho \rangle$ and $\langle c, \sigma \rangle \to \langle \sigma, \rho \rangle$ where $\rho$ is the set of database values that were loaded during the entire computation. S-TRAVERSE is the only rule that loads database values, so the rule adds the newly loaded values $\rho_f$ to the set.

All other rules are modified to collect the loaded values for any sub-computation, where $\bigcup \rho_i$ is shorthand for $\bigcup_{i \in \{1,\ldots,n\}} \rho_i$. Evaluating our running example with the extended semantics generates the following set of database values:

$$\{\mathsf{r}_0, \mathsf{r}_1, 50000, \mathsf{r}_2, 60000, \mathsf{r}_3, 70000, \text{“Cathy”}\}$$

## 3. Analyzing Traversals

Traversal analysis is an abstract interpretation [13] of the operational semantics in which database values are replaced by paths. The *path* corresponding to a database value is the sequence of field names traversed to load that value. Note that many database values may have the same path; for example, in Fig. 4 the paths to record identifiers $\mathsf{r}_1$, $\mathsf{r}_2$, and $\mathsf{r}_3$ are identical. Many paths may lead to the same database value; for example, the value “Bob” can be reached by following either $\mathsf{employees}.\iota.\mathsf{name}$ or $\mathsf{employees}.\iota.\mathsf{manager}.\mathsf{name}$.

Abstract interpretation uses abstraction and concretization functions to specify the relationship between abstract and concrete values. Given an abstract path, its concretization is the set of database values that can be reached by following the path. If the path includes a collection field, the concretized result includes all the traversals of items in the collection. Thus concretization corresponds to interpreting the path as a query against the database.

The analysis is conservative and sound, so that the concretization of a path may return a larger set of database values than the concrete execution of the program actually loads. However, because a program typically operates over a small subset of a large database, the amount of data represented by the concretized paths should be small relative to the overall database size. Soundness justifies bulk loading of data before executing the program; precision gives better performance. The analysis in this section uses a loose

$$\langle l, \sigma\rangle \to \langle \sigma[l], \emptyset\rangle \quad \text{(U-VAR)}$$

$$\langle \textbf{skip}, \sigma\rangle \to \langle \sigma, \emptyset\rangle \quad \text{(U-SKIP)}$$

$$\frac{\langle e, \sigma\rangle \to \langle r, \rho_e\rangle \quad \rho_f = \mathit{Load}(r, f)}{\langle e.f, \sigma\rangle \to \langle \rho_f, \rho_e \cup \rho_f\rangle} \quad \text{(U-TRAVERSE)}$$

$$\frac{\langle e_i, \sigma\rangle \to \langle v_i, \rho_i\rangle \quad \textbf{for } i \in \{1, \ldots, n\}}{\langle \mathsf{op}_n(e_1, \ldots, e_n), \sigma\rangle \to \langle f_{\mathsf{op}_n}(v_1, \ldots, v_n), \cup\rho_i\rangle} \quad \text{(U-OP)}$$

$$\frac{\langle c_1, \sigma\rangle \to \langle \sigma', \rho_1\rangle \qquad \langle c_2, \sigma'\rangle \to \langle \sigma'', \rho_2\rangle}{\langle c_1; c_2, \sigma\rangle \to \langle \sigma'', \rho_1 \cup \rho_2\rangle} \quad \text{(U-SEQ)}$$

$$\frac{\langle e, \sigma\rangle \to \langle v, \rho_e\rangle}{\langle l{:=}e, \sigma\rangle \to \langle [l \mapsto v]\sigma, \rho_e\rangle} \quad \text{(U-ASSIGN)}$$

$$\frac{\langle e, \sigma\rangle \to \langle \mathit{true}, \rho_e\rangle \qquad \langle c_1, \sigma\rangle \to \langle \sigma', \rho_c\rangle}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle \sigma', \rho_e \cup \rho_c\rangle} \quad \text{(U-IFT)}$$

$$\frac{\langle e, \sigma\rangle \to \langle \mathit{false}, \rho_e\rangle \qquad \langle c_2, \sigma\rangle \to \langle \sigma', \rho_c\rangle}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle \sigma', \rho_e \cup \rho_c\rangle} \quad \text{(U-IFF)}$$

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \to \langle \{r_1, \ldots, r_n\}, \rho_1\rangle \\ \langle c, [l \mapsto r_i]\sigma_i\rangle \to \langle \sigma_{i+1}, \rho_{i+1}\rangle \quad \textbf{for } i \in \{1, \ldots, n\}\end{array}}{\langle \textbf{for } l \textbf{ in } e \textbf{ do } c, \sigma_1\rangle \to \langle \sigma_{n+1} \backslash l, \cup\rho_i\rangle} \quad \text{(U-FOR)}$$

**Figure 6.** Operational semantics, extended to collect used-sets.

$$\langle l, \hat\sigma\rangle \hat\to \langle \hat\sigma[l], \emptyset\rangle \quad \text{(A-VAR)}$$

$$\pi_f = \begin{cases} \top & f^t \in \pi_e \\ \{p.f^t \mid p \in \pi_e\} & \text{otherwise} \end{cases}$$
$$\frac{\langle e, \hat\sigma\rangle \hat\to \langle \pi_e, \pi\rangle}{\langle e.f^t, \hat\sigma\rangle \hat\to \langle \pi_f, \pi \sqcup \pi_f\rangle} \quad \text{(A-TRAVERSE)}$$

$$\frac{\langle e, \hat\sigma\rangle \hat\to \langle \hat v, \pi\rangle}{\langle l{:=}e, \hat\sigma\rangle \hat\to \langle [l \mapsto \hat v] \sqcup \hat\sigma, \pi\rangle} \quad \text{(A-ASSIGN)}$$

$$\frac{\langle e_i, \hat\sigma\rangle \hat\to \langle \hat v_i, \pi_i\rangle \qquad \textbf{for } i \in \{1, \ldots, n\}}{\langle \mathsf{op}_n(e_1, \ldots, e_n), \hat\sigma\rangle \hat\to \langle \top, \bigsqcup \pi_i\rangle} \quad \text{(A-OP)}$$

$$\frac{\begin{array}{c}\langle e, \hat\sigma\rangle \hat\to \langle \pi_e, \pi\rangle \\ \pi_\iota = \{p.\iota^l \mid p \in \pi_e\} \\ (\hat\sigma', \pi') = \bigsqcup\{(\textbf{do}\langle c, l, \pi_\iota\rangle)^n(\hat\sigma, \emptyset) \mid n \in \mathbb{N}\}\end{array}}{\langle \textbf{for } l \textbf{ in } e \textbf{ do } c, \hat\sigma\rangle \hat\to \langle \hat\sigma' \backslash l, \pi \sqcup \pi_\iota \sqcup \pi'\rangle} \quad \text{(A-FOR)}$$

$$\frac{\begin{array}{c}\langle e, \hat\sigma\rangle \hat\to \langle \hat v, \pi_e\rangle \\ \langle c_1, \hat\sigma\rangle \hat\to \langle \hat\sigma_1, \pi_1\rangle \qquad \langle c_2, \hat\sigma\rangle \hat\to \langle \hat\sigma_2, \pi_2\rangle\end{array}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat\sigma\rangle \hat\to \langle \hat\sigma_1 \sqcup \hat\sigma_2, \pi_e \sqcup \pi_1 \sqcup \pi_2\rangle} \quad \text{(A-IF)}$$

$$\frac{\langle c, [l \mapsto \pi_\iota] \sqcup \hat\sigma\rangle \hat\to \langle \hat\sigma', \pi'\rangle}{\textbf{do}\langle c, l, \pi_\iota\rangle(\hat\sigma, \pi) = (\hat\sigma', \pi \sqcup \pi')} \quad \text{(A-DO)}$$

**Figure 7.** Path-based abstract interpretation for approximating used-sets.

approximation, but serves as a useful foundation for the more precise analysis in Section 4.

### 3.1 Abstract Value Domain

The abstract value domain is

$$\hat v \in \widehat{\mathit{Value}} = \wp(\mathit{Path}) + \top$$

where $\wp$ is the powerset operator and $\mathit{Path}$ is the set of paths a program can traverse. This version of the analysis focuses only on paths; all non-path values (e.g., constants) are abstracted away as $\top$.

The domain forms a potentially infinite, complete lattice ordered by the subset relation ($\subseteq$). To ensure that the analysis terminates, we force the lattice to be finite by restricting $\mathit{Path}$ to be a finite subset of the all possible sequences of fields $\mathit{Field}^*$. One possible finite subset is the set of paths in which each field name occurs at most once—any other paths would be abstracted as $\top$. With this domain, the expression root.manager.manager would be assigned abstract value $\top$, even though it is a finite path. On the other hand, in the following program x should be assigned value $\top$, because it produces a path of unbounded length:

```
for employee in root.employees do
  x := x.manager;
```

Note that this program is not very useful because there is usually not a meaningful relationship between the number of employees in a list and the depth of a manager traversal.

To distinguish these cases, we create the domain $\mathit{Path}$ by labeling each field in the program, and considering all paths in which each labeled field occurs at most once. With labels, the first expression $\mathsf{root.manager}^1.\mathsf{manager}^2$ has a finite path, but x in the example above would still be assigned abstract value $\top$. More expressive abstractions for representing infinite paths would certainly be useful, for example in the analysis of recursive procedures. Section 6 discusses a more sophisticated alternative; however, such abstractions are beyond the scope of the current work.

### 3.2 Abstract Semantics for Traversals

The operational semantics in Fig. 7 computes the paths a program may traverse. For brevity we omit the rules A-SKIP and A-SEQ, which merely collect values and paths for subcomputations.

The abstract semantics has evaluation relations for expressions $\langle e, \hat\sigma\rangle \hat\to \langle \hat v, \pi\rangle$ and commands $\langle c, \hat\sigma\rangle \hat\to \langle \hat\sigma', \pi\rangle$, where $\hat v$ is an abstract value, $\hat\sigma$ maps variables to abstract values, and $\pi$ is the set of paths traversed by a computation.

Rule A-TRAVERSE defines how field traversal extends a path. In evaluating $e.f^t$, if $e$ yields the set of paths $\pi_e$ then the result of the traversal extends each path in $\pi_e$ with the labeled field $f^t$. The traversal rule also includes a widening clause to ensure the analysis converges. If the field label $t$ already appears in one of the paths that $e$ may traverse, then the program traverses an invalid path in the sense described by Section 3.1. In this case, the analysis approximates the expression's traversals with $\top$.

Rule A-OP gives $\top$ as the abstract value for any operation, because the analysis ignores basic values. In Section 4 we extend the analysis to include abstractions for basic values.

Rule A-IF combines the paths traversed in evaluating the condition and the two command branches of an **if** statement. The join $\hat{\sigma}_1 \sqcup \hat{\sigma}_2$ of two maps $\hat{\sigma}_1$ and $\hat{\sigma}_2$ is a map that includes all elements of both maps:

$$(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)[l] \quad = \quad \hat{\sigma}_1[l] \sqcup \hat{\sigma}_2[l]$$

If $\hat{\sigma}_i$ is not defined for $l$, then $\hat{\sigma}_i[l] = \emptyset$.

Rule A-VAR retrieves a variable's abstract value from the store and does not generate any new paths.

Rule A-ASSIGN describes how a program binds a variable to an abstract value. To ensure soundness, the store maintains a may-be-bound-to relationship between variables and abstract values. Thus the binding operation is a join, rather than an overwrite.

Rule A-FOR evaluates the expression $e$ to determine the paths $\pi_e$ representing the possible collections to be iterated. The rule generates a set of possible paths $\pi_\iota$ to which the loop variable may refer by appending to $\pi_e$ the iterator field name $\iota^l$. These loop variable paths stand for a particular element of the collection. Thus, if the path to the collection is $f_1.f_2$ and the collection's elements each has a field $f_3$, then the path to one of the element's $f_3$ field is $f_1.f_2.\iota^l.f_3$. Each collection iteration (**for** loop) that appears in a program has a unique iterator field name $\iota^l$, where $l$ is the corresponding loop variable. Iterator field names are used in transforming the analysis results into a query, as discussed in Section 5.

The analysis approximates the loop body's concrete behavior by taking the transitive closure of abstractly executing the loop an arbitrary number of times. This value is the least upper bound of a function **do** that is specialized for a given command $c$, loop variable $l$, and set of loop variable paths $\pi_\iota$. The function takes an initial store $\hat{\sigma}$ and set of paths $\pi$ and evaluates $c$ under an updated store that maps $l$ to $\pi_\iota$ to yield a new store $\hat{\sigma}'$ and a new set of paths $\pi'$. The result of the function is $\hat{\sigma}'$ and the combined path set $\pi \sqcup \pi'$.

The abstract evaluation of our running example generates a final store with the following mappings:

$$\mathsf{root} \mapsto \{\epsilon\},\ \textbf{output} \mapsto \top$$

and generates the following set of paths:

$$\{\ \epsilon,\ \mathsf{employees},\ \mathsf{employees}.\iota^e,\ \mathsf{employees}.\iota^e.\mathsf{salary},\ \mathsf{employees}.\iota^e.\mathsf{name},\ \mathsf{employees}.\iota^e.\mathsf{manager},\ \mathsf{employees}.\iota^e.\mathsf{manager.name}\ \}$$

### 3.3 Soundness

The analysis is sound if it safely approximates the values a program loads. If the database stores a set of values $V$, and if executing a program causes the set of persistent values $\rho \subseteq V$ to be loaded, then the analysis should describe a set of values $\hat{\rho}$ such that $\rho \subseteq \hat{\rho} \subseteq V$. We formalize this relation between concrete and abstract values and show that the operational semantics preserves the relation.

The set of concrete values described by an abstract path is the set of values reachable by following that path from the root. We can formalize this description by lifting the definition of $Load$ to operate on paths:

$$Load(r, \epsilon) = \{r\} \qquad \text{(P-LOAD}_1\text{)}$$

$$\frac{Load(r, f) = \{r_1, \ldots, r_n\}}{Load(r, f.p) = \bigcup Load(r_i, p)} \qquad \text{(P-LOAD}_2\text{)}$$

$$Load(r, \iota^l.p) = Load(r, p) \qquad \text{(P-LOAD}_3\text{)}$$

Rule P-LOAD$_1$ states that traversing an empty path from record identifier $r$ yields the set containing $r$. Rule P-LOAD$_2$ loads one level of the traversal hierarchy, then recursively loads the remainder of the hierarchy. Rule P-LOAD$_3$ removes an iterator field name from a path, essentially binding the name to record identifier $r$.

A set of paths $\pi$ safely approximates a set of values $\rho$ if the set of values reachable by following all paths in $\pi$ is a superset of $\rho$:

$$\rho\ \mathcal{R}\ \pi \quad \Leftrightarrow \quad \rho \subseteq \bigcup_{p \in \pi} Load(\mathsf{r}_0, p)$$

For our running example, $\bigcup_{p \in \pi} Load(\mathsf{r}_0, p) =$

$$\{\ \mathsf{r}_0,\ \mathsf{r}_1,\ 50000,\ \text{“Adam”},\ \mathsf{r}_2,\ 60000,\ \text{“Bob”},\ \mathsf{r}_3,\ 70000,\ \text{“Cathy”}\ \}$$

which safely approximates the concrete results.

The abstract domain consists of sets of paths and $\top$, so we lift $\mathcal{R}$ to relate a concrete value $v$ to an abstract value $\hat{v}$ as follows:

$$v\ \mathcal{R}\ \hat{v} \quad \Leftrightarrow \quad \begin{cases} \{v\}\ \mathcal{R}\ \hat{v} & v = r, \hat{v} = \pi \\ v\ \mathcal{R}\ \hat{v} & v = \{r_1, \ldots, r_n\}, \hat{v} = \pi \\ \text{true} & \hat{v} = \top \end{cases}$$

The first two cases relate record identifiers and paths, as above. The final case states that $\top$ always safely approximates a basic program value. We lift $\mathcal{R}$ to relate stores as follows:

$$\sigma\ \mathcal{R}\ \hat{\sigma} \quad \Leftrightarrow \quad \forall x \in Dom(\sigma) \cap Dom(\hat{\sigma}).\sigma[x]\ \mathcal{R}\ \hat{\sigma}[x]$$

By these definitions, the initial stores are *compatible*, in that the initial abstract store safely approximates the initial concrete store.

We define an ordering on abstract stores as follows:

$$\begin{aligned} \hat{\sigma}_1 \sqsubseteq \hat{\sigma}_2 \quad &\Leftrightarrow \quad Dom(\hat{\sigma}_1) \subseteq Dom(\hat{\sigma}_2) \\ &\wedge \quad \forall x \in Dom(\hat{\sigma}_1) \cap Dom(\hat{\sigma}_2).\hat{\sigma}_1[x] \sqsubseteq \hat{\sigma}_2[x] \end{aligned}$$

The abstract semantics rules must be monotone with respect to this ordering, to ensure fixpoint convergence. Conceptually, the rules' monotonicity is evident because the rules never discard a path, assign the value $\top$ to non-path-based expressions, and combine the results of variable assignment. A standard induction over the rules in Fig. 7 formalizes this argument; for brevity, we omit its details.

**Theorem 1 (Soundness of expression evaluation).** *For all $\sigma$, $\hat{\sigma}$, $e$,*

$$\frac{\langle e, \sigma\rangle \to \langle v, \rho\rangle \qquad \langle e, \hat{\sigma}\rangle \to \langle \hat{v}, \pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{(v, \rho)\ \mathcal{R}\ (\hat{v}, \pi)}$$

*Proof.* The proof is a straightforward induction on the structure of $e$. The only interesting case is for rule A-TRAVERSE, when the rule gives $\pi_f = \{p.f^t \mid p \in \pi_e\}$, where $\pi_e$ are the paths $e$ traverses. In this case, the induction hypothesis states that the subexpression paths $\pi_e$ relate to database values $\rho_e$. A simple inductive argument on P-LOAD guarantees that the extended paths $\pi_f$ safely approximate the values the program loads by traversing field $f$ for each value in $\rho_e$. □

**Theorem 2 (Soundness of command evaluation).** *For all $\sigma$, $\hat{\sigma}$, $c$,*

$$\frac{\langle c, \sigma\rangle \to \langle \sigma', \rho\rangle \qquad \langle c, \hat{\sigma}\rangle \to \langle \hat{\sigma}', \pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{(\sigma', \rho)\ \mathcal{R}\ (\hat{\sigma}', \pi)}$$

*Proof.* By induction on the structure of $c$. Rule A-ASSIGN computes a safe approximation of the store by joining stores. Rule A-IF computes a safe approximation of the program behavior by combining the results of the statement's two branches. The proof of soundness for rule A-FOR makes the natural reliance on the finite domain lattice and the monotonicity of the abstract semantics rules to ensure the existence of a safe fixpoint. □

The analysis in this section is quite imprecise and can therefore lead to an excessive over-approximation of the database values a program requires. For example, the analysis conservatively estimates that the program in Fig. 1 needs the name field for every employee even though the program traverses the name field only if the employee's salary is greater than $65,000.

## 4. Analyzing Traversal Conditions

The precision of the analysis can be significantly increased by considering the conditions under which a program traverses data paths. If a program condition can be expressed in a query language, then the analysis can incorporate that condition in the traversal summary. In this section we describe a class of program conditions expressible in our chosen query language, OQL. We discuss how traversals affect program data dependences, which inform the analysis of persistent values the program must load. We then show how to extend the analysis to handle query conditions and data dependence, and we prove the extension's soundness.

### 4.1 Query Conditions

A *query condition* is a conditional program expression that can be expressed as part of an explicit query and evaluated by a database. Not all program conditions are query conditions. For example, if a program condition contains an operation that the database cannot evaluate (e.g., a test for the presence of a local file), then the program condition cannot be expressed in a query. Thus, the database must be able to evaluate all operations that appear in the condition. The database evaluation should produce the same result as program evaluation.

Databases typically only allow conditions that operate on individual records of a collection; for example, the **select** operator in relational algebra evaluates a condition separately for each tuple in a relation. A program condition can introduce a relation between two different collection elements through either: 1) a loop-carried dependence [1] or 2) multiple iterations over the same collection.

If a program condition contains a loop-carried dependence, then that condition depends upon two values from the same collection and is therefore not a query condition. A loop-carried dependence occurs when the evaluation of an expression in a loop depends upon variables assigned in previous iterations of a loop. Figure 8 (a) and (b) contain examples, where the notation C[I] means that condition C can depend upon variable I. The condition in (a) is a query condition because x is redefined in each iteration of the loop. The condition in (b) is not because x has a loop-carried dependence.

Aggregations (counts or sums) necessarily involve a loop-carried dependence; however, these aggregations are typically not used in conditions inside the loop. Note that the current analysis does not convert aggregations into queries, but leaves them as procedural code.

If a program condition depends on loop variables from different iterations of the same collection, then it is not a query condition. If a program condition depends only on the loop variable for the iteration in which the condition appears, then that condition's paths are *distinct*. Both conditions of example (c) have distinct paths and are query conditions. Condition $C_2$ of example (d) is not a query condition, because it depends on a loop variable bound in a different iteration. Note that the subset of OQL the analysis employs can return only one collection for path p, so the conditions from example (b) would be merged in the query as $C_1 \vee C_2$.

The requirements that a query condition contain distinct paths and be free of loop-carried dependences means that query conditions can refer to no more than one element from a given collection. These requirements are related to the requirements for parallelizing code in a parallelizing compiler [1, 33].

```
for I1 in p
  x := E[I1]
  if C[I1,x] then S
```
(a) C is a query condition

```
for I1 in p
  x := E[I1] + x
  if C[I1,x] then S
```
(b) C is not a query condition

```
for I1 in p
  if C1[I1] then S1
for I2 in p
  if C2[I2] then S2
```
(c) $C_{1,2}$ are query conditions

```
for I1 in p
  if C1[I1] then
    x := e[I1]
for I2 in p
  if C2[I2,x] then S
```
(d) $C_2$ is not a query condition

```
for I1 in p
  for I2 in I1.f
    if C[I1,I2] then S
```
(e) C is a query condition

```
for I1 in p1
  for I2 in p2
    if C[I1,I2] then S
```
(f) C is not a query condition

**Figure 8.** Examples of conditions and iterations.

A query condition may contain paths that refer to elements of more than one collection. However, those elements must be structurally related in the database, because the analysis creates a query whose structure mirrors the database. Thus a program condition that appears in a nested iteration is a query condition only if each nested loop iterates over a path that extends the path of its outer loop(s). Example (e) is a query condition because the expression depends only on paths that satisfy this requirement. Example (f) is not a query condition because the inner loop does not extend the path of the outer loop.

Although these requirements restrict the program conditions that may characterize data traversals, we believe these restrictions support common programming idioms used in data-intensive applications. A more powerful query translation could support even more complex conditions.

### 4.2 Data Dependences

A program's data dependences [1] provide information about which persistent values the program must retrieve. If a persistent value affects the contents of the final store, the program must retrieve that value. Assignment statements introduce data dependences, because any assigned value may affect the contents of the final store. Loop variables, however, do not directly induce a data dependence on the final store, because these variables are removed from the store after the loop terminates. Program conditions also do not introduce data dependence, because conditional expressions cannot modify the store. We extend our analysis to collect information about which paths induce data dependences. The query creation algorithm in Section 5 uses this information to ensure retrieval of all values represented by data-dependent paths.

### 4.3 Domains for Paths with Conditions

A *conditional path* $p[k]$ represents a query of the database for values located at path $p$ for which the condition $k$ is true. The condition is expressed as an operation on abstract values, including other paths. The domain of abstract values is extended to include conditions:

$$\begin{aligned} k \in \mathit{Condition} &::= \mathsf{op}_n^t(\hat{v}_1, \ldots, \hat{v}_n) \\ cp \in \mathit{CPath} &::= p[k] \mid p[k]^{\downarrow} \\ \hat{v} \in \widehat{\mathit{Value}} &::= \wp(\mathit{CPath}) + \wp(\mathit{Condition}) + \top \end{aligned}$$

$$\frac{\begin{array}{c}e \text{ contains no loop-carried dependences}\\ k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \pi_e\rangle \\ Distinct(Paths(\hat{v})) \qquad Trim(Paths(\hat{v})) \subseteq I \\ (k \wedge \hat{v}), I \vdash \langle c_1, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}_1, \pi_1\rangle \\ (k \wedge \neg\hat{v}), I \vdash \langle c_2, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}_2, \pi_2\rangle \\ \pi' = \pi_e \sqcup \pi_1 \sqcup \pi_2\end{array}}{k, I \vdash \langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}_1 \sqcup \hat{\sigma}_2, \pi'\rangle} \quad \text{(K-IF}_1\text{)}$$

$$\frac{\begin{array}{c}k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \pi_e, \pi\rangle \\ \pi_f = \begin{cases} \top & f^t \in \pi_e \\ \{p.f^t[k] \mid p \in \pi_e\} & f^t \notin \pi_e \end{cases}\end{array}}{k, I \vdash \langle e.f^t, \hat{\sigma}\rangle \hat{\rightarrow} \langle \pi_f, \pi \sqcup \pi_f\rangle} \quad \text{(K-TRAVERSE)}$$

$$\frac{\begin{array}{c}\text{K-IF}_1 \text{ does not apply} \\ k, I \vdash \langle c_1, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}_1, \pi_1\rangle \\ k, I \vdash \langle c_2, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}_2, \pi_2\rangle \\ \pi' = \pi_e \sqcup \pi_1 \sqcup \pi_2\end{array}}{k, I \vdash \langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}_1 \sqcup \hat{\sigma}_2, \pi'\rangle} \quad \text{(K-IF}_2\text{)}$$

$$\frac{k, I \vdash \langle e_i, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}_i, \pi_i\rangle \qquad \textbf{for } i \in \{1, \ldots, n\} \qquad \hat{v} = \begin{cases} \top & \mathsf{op}_n^t \in \hat{v}_i \\ \mathsf{op}_n^t(\hat{v}_1, \ldots, \hat{v}_n) & \mathsf{op}_n^t \notin \hat{v}_i \end{cases}}{k, I \vdash \langle \mathsf{op}_n^t(e_1, \ldots, e_n), \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \bigsqcup \pi_i\rangle} \quad \text{(K-OP)}$$

$$\frac{k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \pi\rangle}{k, I \vdash \langle l{:=}e, \hat{\sigma}\rangle \hat{\rightarrow} \langle [l \mapsto \hat{v}] \sqcup \hat{\sigma}, \pi^{\downarrow} \sqcup I_{|I|}[k]^{\downarrow}\rangle} \quad \text{(K-ASSIGN)}$$

$$\frac{\begin{array}{c}k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \pi_e, \pi\rangle \\ \pi_\iota = \{p.\iota^l \mid p \in \pi_e\} \\ I' = Extend(I, \pi_\iota) \\ (\hat{\sigma}', \pi') = \bigsqcup\{(\textbf{do}\langle k, I', c, l, \pi_\iota\rangle)^n(\hat{\sigma}, \emptyset) \mid n \in \mathbb{N}\}\end{array}}{k, I \vdash \langle \textbf{for } l \textbf{ in } e \textbf{ do } c, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}' \backslash l, \pi \sqcup \pi_\iota \sqcup \pi'\rangle} \quad \text{(K-FOR)}$$

$$\frac{\forall p_1, p_2 \in \pi : Erase(p_1) = Erase(p_2) \Rightarrow p_1 = p_2}{Distinct(\pi)}$$

$$\frac{k, I \vdash \langle c, [l \mapsto \pi_\iota] \sqcup \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}', \pi'\rangle}{\textbf{do}\langle k, I, c, l, \pi_\iota\rangle(\hat{\sigma}, \pi) = (\hat{\sigma}', \pi \sqcup \pi')} \quad \text{(K-DO)}$$

$$\begin{array}{rcl} Erase(\bar{f}_1.\iota^{l_1}. \cdots .\iota^{l_n}.\bar{f}_{n+1}) & = & \bar{f}_1. \cdots .\bar{f}_{n+1} \\ Trim(\pi) & = & \{p.\iota^l \mid p.\iota^l.\bar{f} \in \pi\} \\ Paths(\pi) & = & \pi \\ Paths(\mathsf{op}_n(\hat{v}_1, \ldots, \hat{v}_n)) & = & Paths(\hat{v}_1) \sqcup \cdots \sqcup Paths(\hat{v}_n) \end{array}$$

$$Extend(I, \pi_\iota) = \begin{cases} \pi_\iota & |I| = 0 \\ I,\ \pi_\iota & Prefixes(I_{|I|}, \pi_\iota) \\ I & \text{otherwise} \end{cases}$$

$$\frac{\forall p_2 \in \pi_2 : (\exists p_1 \in \pi_1, \exists p' \in Field^* : p_1.p' = p_2)}{Prefixes(\pi_1, \pi_2)}$$

**Figure 9.** Abstract interpretation with paths and conditions.

A path marked $p[k]^{\downarrow}$ is involved in a data dependence. A non-conditional path $p$ is lifted to a conditional path $p[true]$ signifying that the path is always traversed. The label $t$ on an operator is analogous to field labels in Section 3.1. Conditions are restricted to include only one occurrence of a labeled operator, allowing the domain $CPath$ to be finite.

### 4.4 Abstract Semantics for Conditional Traversals

Figure 9 extends the abstract semantics to include the condition under which a program traverses a path. The evaluation relation $k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \pi\rangle$ now carries a context that consists of the condition $k$ under which traversals may take place and the *collection traversal list* $I$. This list represents the nesting structure of collection traversals and is used to identify query conditions. The first element of the list is the set of paths for the outermost loop variable, and the last element is the set of paths for the innermost (current) loop variable. If list $I$ has length $|I|$, then $I_{|I|}$ denotes the innermost loop variable paths. Initially $k$ is set to $true$, and $I$ is empty.

Our discussion of the extended semantics highlights the additions necessary to identify and attach query conditions, according to the requirements described in Sections 4.1 and 4.2. These additions generally either use or modify the context. We omit rules K-VAR, K-SKIP, and K-SEQ because these rules only collect the results of subcomputation and do not directly alter the context.

Rule K-IF$_1$ identifies query conditions. The analysis first determines that $e$ contains no loop-carried dependences. This determination is the result of a standard analysis; for brevity, we omit its details.

The premise $Distinct(Paths(\hat{v}))$ ensures that all the paths in the condition's abstract value are *distinct*, in the sense that they do not traverse the same set of fields with different iteration field names.

The analysis also checks that all the paths used in the expression are based on lexically enclosing iteration paths. The function $Trim$ applied to a set of paths $\pi$ returns all possible paths for the inner-most loop variable. Thus, the premise $Trim(Paths(\hat{v})) \subseteq I$ ensures that any iteration paths that appear in the condition's abstract value are based on lexically enclosing iteration paths.

The lexically enclosing iterations needed by K-IF$_1$ are created by K-FOR. The rule appends a new set of paths $\pi_\iota$ to the list of iteration paths $I$ only if all paths in $\pi_\iota$ extend some path in $I_{|I|}$, the list's most recently added member. In this way, K-FOR maintains the constraint that $I$ is a list of lexically enclosing iteration paths. Other than imposing this constraint on $I$, K-FOR is the same as A-FOR (Fig. 7).

Query conditions are used in the true and false branches of the **if** command. Given the abstract value $\hat{v}$ of condition $e$, the true-branch body is evaluated under the condition $k \wedge \hat{v}$ and the false-branch body under the condition $k \wedge \neg\hat{v}$. When the program makes a traversal, rule K-TRAVERSE attaches the condition $k$ to the path generated by the traversal.

If the condition does not satisfy the requirements of a query condition, rule K-IF$_2$ does not augment the paths with conditions.

Rule K-ASSIGN performs assignments but also marks all paths in the bound expression as having data dependences. $\pi^{\downarrow}$ means the marking of all paths in $\pi$ with $^{\downarrow}$, and $I_{|I|}[k]^{\downarrow}$ means marking all paths in $I_{|I|}$ with condition $k$ and $^{\downarrow}$. The loop variable paths themselves are marked as a data dependence because the execution of any assignment can depend upon the *existence* of an element in an iteration, even if no fields of the loop variable are used. For

example, in the program **for** x **in** p **do** y := y + 1, the variable x is never used, yet there is still a data dependence upon it because its elements must be enumerated.

Rule K-OP defines the semantics of operations on abstract values. The operands are evaluated and the operator is retained in the result. The rule also includes a widening clause to ensure convergence to a fixed-point. The rule is similar to the one for traversals: If the syntactic use of the operator $\mathsf{op}_n$ already occurs in some $\hat{v}_i$, then the expression evaluates to $\top$.

The abstract evaluation of our running example generates a final store with the following mappings:

$$\begin{array}{l}\mathsf{root} \mapsto \{\epsilon\},\\ \textbf{output} \mapsto \{print(\mathsf{employees}.\iota^e.\mathsf{name}[k]+\\ \qquad\qquad \mathsf{employees}.\iota^e.\mathsf{manager.name}[k])\}\end{array}$$

and the following set of paths:

$$\begin{array}{l}\{\ \epsilon,\ \mathsf{employees},\ \mathsf{employees}.\iota^e,\ \ \mathsf{employees}.\iota^e.\mathsf{salary},\\ \mathsf{employees}.\iota^e.\mathsf{name}[k],\ \mathsf{employees}.\iota^e.\mathsf{manager}[k],\\ \mathsf{employees}.\iota^e.\mathsf{manager.name}[k],\ \mathsf{employees}.\iota^e[k]^{\downarrow},\\ \mathsf{employees}.\iota^e.\mathsf{name}[k]^{\downarrow},\ \mathsf{employees}.\iota^e.\mathsf{manager}[k]^{\downarrow},\\ \mathsf{employees}.\iota^e.\mathsf{manager.name}[k]^{\downarrow}\ \}\end{array}$$

where $k$ is $\{\mathsf{employees}.\iota^e.\mathsf{salary}\} > 65000$.

At this stage of the analysis, the conditions apply to the final attributes loaded by a path. In Section 5 we further analyze the conditions and paths to avoid loading entire records.

### 4.5 Soundness

Proceeding as before, we define the load operation for conditional paths and define the relations between concrete and abstract domains. We then prove that evaluation preserves these relations. This proof relies on the previous soundness proof (Section 3.3).

The load operation for conditional paths is defined in Fig. 10. Function $CLoad_i$ loads all records reachable by a path $p$, provided the path's condition $k$ may be $true$. A mapping $\phi$ binds an iterator field name to a specific record identifier, to be referenced in the evaluation of the path's condition. $CLoad_1$ creates iterator field name bindings, and $CLoad_2$ uses the bindings.

Function $eval$ defines condition evaluation. Operator evaluation calls $eval$ on the operands and applies $f'_{\mathsf{op}_n}$ to the results, where

$$f'_{\mathsf{op}_n}(\hat{v}_1,\ldots,\hat{v}_n) = \begin{cases}\top & \top \in \{\hat{v}_1,\ldots,\hat{v}_n\}\\ f_{\mathsf{op}_n}(\hat{v}_1,\ldots,\hat{v}_n) & \text{otherwise}\end{cases}$$

Note that because the underlying operators are monotonic, all functions $f'$ are also monotonic.

Path evaluation calls $CLoad_2$ on the path, passing bindings $\phi$ for any iterator field names that appear in the path. Note that, because the evaluated path can appear only in an operation expression, the result of path evaluation must be a set that contains a single basic value. The least upper bound operation ($\bigsqcup$) retrieves this value from the set. Evaluating a set of abstract values yields the least upper bound of evaluating each value in the set.

A set $\pi$ of conditional paths safely approximates the persistent values a program loads if it describes a superset of those values. We modify the definitions of $\mathcal{R}$ to relate concrete values and conditional paths:

$$\begin{array}{rcl}
CLoad_i(r,\epsilon[k],\phi) & = & \begin{cases}\{r\} & true \sqsubseteq eval(k,\phi)\\ \emptyset & \text{otherwise}\end{cases}\\
CLoad_i(r,f.p[k],\phi) & = & \bigcup_{r'\in Load(r,f)} CLoad_i(r',p[k],\phi)\\
CLoad_1(r,\iota^l.p[k],\phi) & = & CLoad_1(r,p[k],[\iota^l\mapsto r]\phi)\\
CLoad_2(r,\iota^l.p[k],\phi) & = & CLoad_2(\phi(\iota^l),p[k],\phi)\\[2ex]
eval(p[k],\phi) & = & \bigsqcup CLoad_2(\mathsf{r}_0,p[k],\phi)\\
eval(\mathsf{op}_n(\hat{v}_1\ldots\hat{v}_n),\phi) & = & f'_{\mathsf{op}_n}(eval(\hat{v}_i,\phi),\ldots,eval(\hat{v}_n,\phi))\\
eval(S,\phi) & = & \bigsqcup_{\hat{v}\in S} eval(\hat{v},\phi)
\end{array}$$

**Figure 10.** Conditional record loading.

$$\begin{array}{rcl}
\rho\,\mathcal{R}\,\pi & \Leftrightarrow & \rho \subseteq \bigcup_{p[k]\in\pi} CLoad_1(\mathsf{r}_0,p[k],\emptyset)\\
(v,\sigma)\,\mathcal{R}\,\hat{v} & \Leftrightarrow & \begin{cases}\{v\}\,\mathcal{R}\,\hat{v} & v=r,\hat{v}=\pi\\ v\,\mathcal{R}\,\hat{v} & v=\{r_1,...,r_n\},\hat{v}=\pi\\ v\sqsubseteq eval(\hat{v},\phi_\sigma) & v\in Basic,\hat{v}=k\\ \hat{v}=\top & \text{otherwise}\end{cases}\\
\sigma\,\mathcal{R}\,\hat{\sigma} & \Leftrightarrow & \forall x\in Dom(\sigma)\cap Dom(\hat{\sigma}).(\sigma[x],\sigma)\,\mathcal{R}\,\hat{\sigma}[x]
\end{array}$$

The relation between concrete and abstract values is defined only in the context of a store, because the store provides a binding for any iterator field names that may appear in paths and conditions. When $\hat{v}$ is an abstract operation, evaluating $\hat{v}$ must approximate $v$. If $L$ is the set of loop variables that appear in the entire program, $\phi_\sigma = \bigcup_{l\in L}[\iota^l \mapsto \sigma[l]]$, where $\sigma[l] = \top$ if $\sigma[l]$ is undefined.

To prove soundness, we first show that expression evaluation gives the same results as the analysis in Section 3.3, *assuming that evaluating every path condition may give the value true*. We then show that the analysis only constructs conditions that satisfy this assumption. Proof of soundness for command evaluation follows trivially.

**Lemma 1 (Subcomputation compatibility).** *If* $(\rho_1\,\mathcal{R}\,\pi_1)$ *and* $(\rho_2\,\mathcal{R}\,\pi_2)$, *then* $(\rho_1\cup\rho_2)\,\mathcal{R}\,(\pi_1\sqcup\pi_2)$.

*Proof.* If $\pi_1\sqcup\pi_2 = \top$, then the relation trivially holds. Otherwise, by the definition of $\mathcal{R}$, $\rho_1 \subseteq \bigcup_{p[k]\in\pi_1} CLoad_1(\mathsf{r}_0,p[k],\emptyset)$ and $\rho_2 \subseteq \bigcup_{p[k]\in\pi_2} CLoad_1(\mathsf{r}_0,p[k],\emptyset)$. Assuming all conditions $k$ may be true, $\rho_1\cup\rho_2 \subseteq \bigcup_{p[k]\in\pi_1\cup\pi_2} CLoad_1(\mathsf{r}_0,p[k],\emptyset)$, so the relation holds. □

**Theorem 3 (Soundness of expression evaluation).** *For all* $e,\sigma,\hat{\sigma},k$:

$$\frac{\langle e,\sigma\rangle\to\langle v,\rho\rangle \qquad k,I\vdash\langle e,\hat{\sigma}\rangle\hat{\to}\langle\hat{v},\pi\rangle \qquad \sigma\,\mathcal{R}\,\hat{\sigma} \qquad true\sqsubseteq eval(k,\phi_\sigma)}{((v,\sigma),\rho)\,\mathcal{R}\,(\hat{v},\pi)}$$

*Proof.* By induction on the structure of $e$.

**Base case** $e\equiv[\![l]\!]$ In this case, $(v,\rho) = (\sigma[l],\emptyset)$ and $(\hat{v},\pi) = (\hat{\sigma}[l],\emptyset)$ The premise $\sigma\,\mathcal{R}\,\hat{\sigma}$ gives the desired result.

The induction hypothesis asserts that evaluating subexpressions under condition $k$ produces sound results. It remains to show that

evaluating operators and traversals under condition $k$ produces sound results.

**Case** $e \equiv [\![\mathsf{op}_n^t(e_1 \ldots e_n)]\!]$ If the abstract semantics gives $\hat{v} = \top$ for $e$, then this case is trivially proved. Otherwise, it must be shown that if, for each $e_i$, the concrete semantics gives $(v_i, \rho_i)$ and the abstract semantics gives $(\hat{v}_i, \pi_i)$, then:

$$\begin{array}{c} f_{\mathsf{op}_n}(v_1, \ldots, v_n) = f'_{\mathsf{op}_n}(v_1, \ldots, v_n) \\ \sqsubseteq \\ eval(\mathsf{op}_n(\hat{v}_1, \ldots, \hat{v}_n), \phi_\sigma) = f'_{\mathsf{op}_n}(eval(\hat{v}_i, \phi_\sigma), \ldots, eval(\hat{v}_n, \phi_\sigma)) \end{array}$$

Because $f'$ is monotonic, it suffices to show that if $(v_i, \sigma)\ \mathcal{R}\ \hat{v}_i$, then $v_i \sqsubseteq eval(\hat{v}_i, \phi_\sigma)$. If $v_i$ is a basic value, then the definition of $\mathcal{R}$ suffices. It remains to be shown that if $v_i$ is a record identifier,

$$v_i \sqsubseteq \bigsqcup_{p[k] \in \hat{v}_i} \left\{ \bigsqcup CLoad_2(\mathsf{r}_0, p[k], \phi_\sigma) \right\}$$

Because $v_i\ \mathcal{R}\ \hat{v}_i$, there exists some paths $\pi' \subseteq \hat{v}_i$ such that $v_i \in CLoad_1(\mathsf{r}_0, p', \emptyset)$, where $p' \in \pi'$. Calling $CLoad_1$ on these paths generates of set of iterator field bindings $\Phi$ that includes $\phi_\sigma$; therefore $r \in \bigcup_{p[k] \in \hat{v}} CLoad_2(\mathsf{r}_0, p[k], \phi_\sigma)$. Hence, the desired result that evaluating all paths in $\hat{v}_i$ with bindings $\phi_\sigma$ approximates $r$. Lemma 1 gives $\bigcup \rho_i\ \mathcal{R}\ \bigsqcup \pi_i$.

**Case** $e \equiv [\![e.f^t]\!]$ Rules U-TRAVERSE and K-TRAVERSE and the induction hypothesis give $(r, \rho_e)\ \mathcal{R}\ (\pi_e, \pi)$ for the subexpression $e$. For the entire expression, the rules give $\rho_f = Load(r, f)$, $\pi_f = \{p.f^t[k] \mid p \in \pi_e\}$. If $true \sqsubseteq eval(k, \phi_\sigma)$, then $\pi_f = \{p.f^t \mid p \in \pi_e\}$. Section 3.3 proved soundness for this case. Lemma 1 gives $(\rho_e \cup \rho_f)\ \mathcal{R}\ (\pi \sqcup \pi_f)$. □

**Theorem 4 (Condition evaluation approximates** $true$**).** *For all* $\sigma$*,* $\hat{\sigma}$*, conditions* $k$ *produced by the analysis:*

$$\frac{\sigma\ \mathcal{R}\ \hat{\sigma}}{true \sqsubseteq eval(k, \phi_\sigma)}$$

*Proof.* By induction on the structure of $k$.

**Base case** $k = true$ Trivial, because $eval(true, \_) = true$.

The induction hypothesis asserts that evaluating subconditions approximates $true$. It remains to prove the theorem for any condition $k'$ the analysis creates.

**Case** $k' \equiv [\![k \wedge \hat{v}]\!]$**,** $\hat{v}$ **is a query condition** In this case, the analysis attaches $k'$ to all paths generated by the true-branch of an **if**. So, it must be shown:

$$\frac{\langle e, \sigma\rangle \rightarrow \langle true, \rho\rangle \qquad k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{true \sqsubseteq eval(k \wedge \hat{v}, \phi_\sigma)}$$

The induction hypothesis states $true \sqsubseteq eval(k, \phi_\sigma)$, so it remains to show $true \sqsubseteq eval(\hat{v}, \phi_\sigma)$. The induction hypothesis also enables the invocation of Theorem 3, which gives $(true, \sigma)\ \mathcal{R}\ \hat{v}$ which is defined to mean $true \sqsubseteq eval(\hat{v}, \phi_\sigma)$.

**Case** $k' \equiv [\![k \wedge \neg\hat{v}]\!]$**,** $\hat{v}$ **is a query condition** In this case, the analysis attaches $k'$ to all paths generated by the false-branch of an **if**. So, it must be shown:

$$\frac{\langle e, \sigma\rangle \rightarrow \langle false, \rho\rangle \qquad k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{true \sqsubseteq eval(k \wedge \neg\hat{v}, \phi_\sigma)}$$

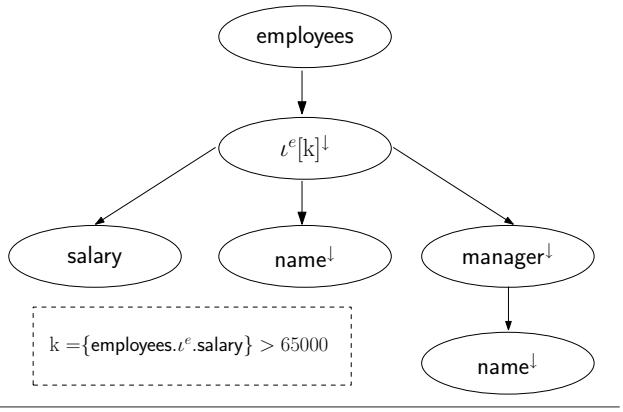


**Figure 11.** Example query tree, with promoted conditions.

Proceeding as above, Theorem 3 gives $false \sqsubseteq eval(\hat{v}, \phi_\sigma)$. Since $f'_\neg$ is monotonic, $true \sqsubseteq \neg eval(\hat{v}, \phi_\sigma)$. A simple analysis on the domain of $f'_\neg$ gives $\neg eval(\hat{v}, \phi) = eval(\neg\hat{v}, \phi)$, and the desired conclusion is reached. □

**Theorem 5 (Soundness of command evaluation).** *For all* $\sigma$*,* $\hat{\sigma}$*, c:*

$$\frac{\langle c, \sigma\rangle \rightarrow \langle \sigma', \rho\rangle \qquad k, I \vdash \langle c, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}', \pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{(\sigma', \rho)\ \mathcal{R}\ (\hat{\sigma}', \pi)}$$

*Proof.* Note that the transfer functions of this extended semantics are also monotonic. Thus the proof of soundness for commands is similar to that of Theorem 2, with appropriate applications of Theorems 3 and 4. □

## 5. Query Creation and Program Simplification

The results of static analysis can now be employed to partition the original program into a query and its client. The query retrieves a subset of the database on which the client program executes. In some cases, the client program may be simplified by removing conditional tests that become redundant when executed on the data subset.

### 5.1 Query Creation

Query creation proceeds as a depth-first traversal of a program's *query tree*—the result of combining all paths discovered by the program's query extraction analysis. If two paths differ only in their conditions, the combined path contains the disjunction of the two conditions; the disjunction represents *all* possible conditions under which the program could traverse that path. If a data dependence marks at least one of two otherwise identical paths, the combined path preserves the data dependence. The preservation indicates that *some* program traversal of this path induces a data dependence.

The concretization of a query tree corresponds to running a query against a database. In its current form, the conditions are associated with the use of individual attributes, yet conditionally loading an attribute is not nearly as useful as conditionally loading an entire record. To avoid loading records, the conditions on individual attributes are *promoted* to apply to iteration fields.

We provide an informal argument, rather than a formal proof, for the validity of promotion. Condition promotion depends on the connection between individual attributes and object loading: An object does not need to be loaded if none of its attributes are needed and if the object does not affect the final outcome of the program. Thus the condition for loading an object is the union (disjunction) of the conditions of all uses of its attributes. In this

way, the conditions on attributes are promoted to be conditions on elements of a collection. If any of the paths has the condition $true$, then all elements of the collection will be loaded.

One important point is the difference between marked and unmarked paths. A marked path $p[k]^{\downarrow}$ is a data dependence of the final state of the program store. An unmarked path—for example, a path traversed by a condition—affects only the program's control flow. Conditions on marked paths are promoted. Conditions on unmarked paths are ignored, because these paths do not directly affect the final store.

Figure 11 contains the query tree for the running example, after condition promotion. The condition $k$ was promoted from the employee's name, manager, and manager.name fields to be a *filter* on the elements of collection employees. The implicit condition true on the employee's name field was not promoted, because traversing that field does not induce a data dependence.

An explicit query is expressed in a variant of the Object Query Language (OQL) [8]. The syntax is:

$$\begin{array}{rcl} q & ::= & \textbf{struct (}\ f_1 = q_1\ , \ldots , f_n = q_n\textbf{)} \\ & | & \textbf{select}\ q\ \textbf{from}\ q\ \textbf{as}\ x\ \textbf{where}\ e \\ & | & x.\bar{f} \end{array}$$

where $f$ names a record field, $\bar{f}$ is a sequence of field names, and $x$ is a variable name. We restrict our use of OQL to queries that return a structural subset of the original database. This mirrors the capabilities of commercial products like Hibernate and EJB [22, 28]. Consideration of other query translations is an area for future research.

Figure 12 builds an explicit query from a set of paths. Function $Q$ combines paths, promotes conditions, and outputs an explicit query. The function takes a path $p$ that represents a common prefix for a set of suffix paths $\pi$; it returns a query for the elements reachable by following each suffix from the given prefix.

In rule Q-PATH, the prefix path *is* the query when the suffix path is empty. Recall that query creation promotes conditions to be filters on collections; thus Q-PATH ignores conditions. The rule also ignores data dependence, because this information is used only for query promotion.

Rule Q-FIELDS handles the case where the suffixes all start with distinct field names $f_i$. The query result is a **struct** where each field name is bound to a sub-query for that field. Each field name's query $q_i$ is constructed by appending $f_i$ to the current prefix.

In rule Q-ITER, all suffixes start with an iterator field name $\iota^l$, and the query result selects elements of the collection to which $\iota^l$ refers. If the suffixes begin with different iterator field names, each field name represents a different iteration of the collection. In this treatment we only consider queries that mirror the structure of the database, so only one collection can be returned for a given multi-valued field. Therefore the rule combines queries for multiple iterations. Q-ITER promotes conditions by identifying all suffix paths marked with a data dependence for which $\iota^l$ is the last iterator field name. The rule strips the conditions from these paths and disjoins them to create the **select** statement's **where** clause. The **select** statement's subquery is the result of calling $Q$ on the combined set of stripped paths and remaining paths $\pi'$.

Function $T$ transforms any paths that may appear in a condition. If a path contains an iterator field name $\iota^l$, $T$ removes the prefix that appears before $\iota^l$. Because this field name must be the last to appear in a path, it will be properly scoped by the **as** clause of the **select** query.

Function $T$ also expands sets of paths that may appear in operations. Therefore $T$ disjoins the cross-product achieved by applying the operation to each possible combination of operands.

$$Q(p[k]^{\downarrow}, \{\epsilon\}) = Q(p[k], \{\epsilon\}) = p \qquad \text{(Q-PATH)}$$

$$\frac{\begin{array}{c}\pi \backslash \epsilon = f_1.\pi_1 \cup \cdots \cup f_n.\pi_n \qquad f_i \text{ distinct} \\ q_i = Q(p.f_i, \pi_i))\end{array}}{Q(p, \pi) = \textbf{struct}\ (f_1 = q_1, \ldots, f_n = q_n)} \qquad \text{(Q-FIELDS)}$$

$$\frac{\begin{array}{c}\pi = \iota.\pi' \cup \{\iota^l.\bar{f}_1[k_1]^{\downarrow}, \ldots, \iota^l.\bar{f}_n[k_n]^{\downarrow}\} \qquad \bar{f}[k]^{\downarrow} \notin \pi' \\ q = Q(\iota^l, \pi' \cup \{\bar{f}_1, \ldots, \bar{f}_n\}) \\ c = T(k_1) \vee \cdots \vee T(k_n)\end{array}}{Q(p, \pi) = \textbf{select}\ q\ \textbf{from}\ p\ \textbf{as}\ l\ \textbf{where}\ c} \qquad \text{(Q-ITER)}$$

$$T(\pi) = \{\iota^l.\bar{f} \mid p.\iota^l.\bar{f} \in \pi\}$$
$$T(\mathsf{op}_n(e_1, \ldots, e_n)) = \bigvee\{\mathsf{op}_n(e'_1, \ldots, e'_n) \mid e'_i \in T(e_i)\}$$

**Figure 12.** Transforming conditional path sets to queries

Given a set of conditional paths $\pi$ extracted from a program, the query for $\pi$ is $Q(\epsilon, \pi)$. For our running example the result is:

```
select struct ( name = e.name, salary = e.salary ,
        manager = struct (name = e.manager.name) )
from Employee as e
where e.salary > 65000;
```

### 5.2 Client and Query Simplification

In the next step of the analysis, the data constraints ensured by the query are used to simplify the program, and consequently the data elements in the result of the query. If the program tests a property of the data which is guaranteed by the query, the program test can be removed. Any data that is only used in such tests can then be removed from the query results.

The following two rules are used to simplify the client program:

$$\frac{\Gamma \vdash \hat{v}(e)}{\Gamma \vdash \langle e \rangle \rightarrow \mathit{true}} \qquad \frac{\Gamma \vdash \neg\hat{v}(e)}{\Gamma \vdash \langle e \rangle \rightarrow \mathit{false}}$$

where the context $\Gamma$ is a set of constraints on the persistent data a query returns. The constraints are obtained by taking the conjunction of all the query's **where** clause conditions. The term $\hat{v}(e)$ refers to the abstract value for $e$ produced by the rules of Fig. 9. If a context *entails* an expression's abstract value—written $\Gamma \vdash \hat{v}(e)$—then the expression can be re-written as $true$. Similarly if the context entails the negation of an expression's abstract value, the expression can be re-written as $false$. Entailment can be determined by a SAT solver. The rules are applied repeatedly until no more reductions are possible. Once the client has been simplified, a further analysis can remove trivial tests and dead code [39].

The query subsequently can be simplified by applying the analysis of Fig. 9 to the new client and composing the results with the original query. The composite query does not retrieve values that appear only in **where** clauses. The partition for the example program in Fig. 1 is:

```
// define an explicit query
String query =
  "select struct (
      name = e.name,
      manager = struct (name = e.manager.name))
   from Employee as e
   where e.salary > 65000";
// execute the query
```

```
List result = executeQuery(query);
for (Employee e : result.list()) {
    // no test required: all elements already satisfy
    // the condition
    print(e.name + ": " + e.manager.name);
}
```

The function executeQuery queries the database and returns a new structure that contains only the data retrieved by the query. The query does not retrieve the employee's salary, and the program does not test for that value. Instead, the query retrieves only employees whose salary is greater than $65,000.

## 6. Related Work

Our path-based approach is similar to research on approximating the shape of pointer data structures [17, 19, 40]. However, we limit ourselves to intraprocedural analysis and focus on the traversal of read-only data structures, not mutation. Our current representation of database paths cannot express sophisticated data traversals. For example, our analysis conservatively represents as $\top$ a recursive traversal of a field from a given root. We could use tree automata to represent paths, similar to the storeless model of [14], where the database is the store. In the future, we plan to evaluate these more sophisticated path representations. A question open for future study is how the analysis may benefit from a more expressive path representation, given that few if any existing query languages can match this expressivity.

Vitenberg et al. describe a path-based abstract interpretation for predicting the persistent values a program may need [38]. Their approach supports runtime improvement of transaction lock scheduling. Kvilekval and Singh use shape analysis to dynamically hoard (prefetch) remote data for mobile clients [24]. Their work reduces the effect of disconnections in mobile computing environments. Ours is a fully static approach that supports program transformation to bulk-load persistent data. Our analysis also differs in that it identifies traversal conditions.

Neubauer and Theimann partition a sequential program run at one location into semantically equivalent, independent, distributed processes [31]. Their approach provides software engineering benefits similar to ours, except for multi-tier applications.

A common technique for integrating programming languages and databases is to make queries first-class values of a programming language. C$^{\#}$ has been extended to incorporate relational constructs and structured data in middle-tier applications [7]. Willis et al. propose extensions to Java to support database-style optimizations for operations on collections of objects [41]. Safe queries describe queries with classes whose instances are translated into a form that can be executed on a remote database [11]. Unlike our proposal, each of these solutions reduce persistent transparency because they require explicit queries to be written in an extended programming language syntax.

The DBPL language [36] and it successor Tycoon [29] explored optimization of search and bulk operations within the framework of orthogonal persistence. Tycoon proposed integrating compiler optimization and database query optimization [18]. Queries that cross modular boundaries were optimized at runtime by dynamic compilation [35]. The languages included explicit syntax for writing queries or bulk operations on either persistent or non-persistent data.

Several researchers have extended object persistence architectures to leverage *traversal context*—access patterns, including paths—to dynamically predict database loads and prefetch the predicted values [6, 21, 23]. Because our work generates queries which could be used in object persistence architectures, the two techniques could be combined to achieve further performance benefits.

## 7. Future Work

While the current analysis provides a unique technique for extracting implicit queries from imperative programs, it contains several restrictions, which we hope to remove or diminish with future work. The imperative language we studied contains no procedures. We are currently extending the analysis to analyze whole programs with behavioral methods and recursive procedures.

To transform complete programs, more work is needed to identify where the analysis should be applied. Currently a new query is created each time the special variable root is used. In some cases it may be more efficient to break a query into parts, so that a result of one query becomes the root of a nested query. Multiple queries could also be used to transform programs in which an outer loop introduces a loop-carried dependence. The expressive power of the target query language also affects these decisions. Other strategies for promoting conditions may also be considered.

The current work analyzes only data retrievals. Future work will extend this analysis to include updates to persistent data. If updates are performed immediately, the resulting aliasing may make it impossible to define a useful transformation for updates. Alternatively, it may be possible to delay the updates until a transaction boundary, at which point all database references must be released.

Employing standard static analyses (e.g., range analysis) can improve the quality of the extracted queries. These analyses should also allow us to identify and extract aggregation and "exists" subqueries.

Key differences between programming languages and database semantics must be overcome to successfully integrate the two domains. In this paper, we identified two artifacts of the database domain—the three-valued logic of *null* values and the implicit ordering of database sets—that must have appropriate analogues in the programming languages domain.

Finally, the technique must be applied to realistic programs to measure the performance of the analysis and effectiveness of the transformation.

## 8. Conclusion

We have formalized a new approach for optimizing transparent persistence. This approach extracts a query from an imperative program, then simplifies the program to operate over the bulk-load query results. This technique promises to combine the software engineering benefits of transparent persistence with the performance benefits of query optimization. We expect the current work to serve as a useful foundation for ongoing research into the long-standing effort to integrate programming languages and databases.

## Acknowledgments

We thank Jens Palsberg for advice on appropriate formalisms. We thank Calvin Lin for helpful discussions. We thank David Schmidt, Kathryn McKinley, Jayadev Misra, Mike Bond, Ben Hardekopf, Ali Ibrahim, David Kitchin, and the anonymous ICALP 2006 and POPL 2007 reviewers for helpful comments on the paper.

## References

[1] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proc. of the Symp. on Compiler Construction (CC)*, pages 233–246, 1984.

[2] M. P. Atkinson. Programming languages and databases. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 408–419. IEEE Computer Society, 1978.

[3] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

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

[5] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design - An Entity-Relationship Approach*. Benjamin Cummings, 1992.

[6] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *The VLDB Journal*, pages 327–338, 1999.

[7] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in c$\omega$. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, pages 287–311, 2005.

[8] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, January 2000.

[9] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. of Symp. on Principles of Database System (PODS)*, pages 34–43, 1998.

[10] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[11] W. R. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, pages 97–106, 2005.

[12] G. Copeland and D. Maier. Making smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 316–325. ACM Press, 1984.

[13] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL)*, pages 178–190, 2002.

[14] A. Deutsch. A storeless model of aliasing and its abstractions using finiterepresentations of right-regular equivalence relations. *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, pages 2–13, 1992.

[15] O. Deux. The O2 system. *Commun. ACM*, 34(10):34–48, 1991.

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

[17] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[18] A. Gawecki and F. Matthes. Integrating query and program optimization using persistent CPS representations. In M. P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*, ESPRIT Basic Research Series, pages 496–501. Springer Verlag, 2000.

[19] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL)*, pages 1–15, 1996.

[20] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, pages 645–654, 2004.

[21] W.-S. Han, Y.-S. Moon, and K.-Y. Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. *Information Sciences*, 152(1):47–61, 2003.

[22] Hibernate reference documentation. `http://www.hibernate.org/hib_docs/v3/reference/en/html`, May 2005.

[23] A. Ibrahim and W. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[24] K. Kvilekval and A. Singh. SPREE: Object prefetching for mobile computers. In *Distributed Objects and Applications (DOA)*, Oct 2004.

[25] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the Intl. Conf. on Management of Data (SIGMOD)*, pages 318–329, 1996.

[26] D. Maier. Representing database programs as objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 377–386. New York, NY, 1990.

[27] D. Maier, J. Stein, A. Otis, and A. Purdy. Developments of an object-oriented DBMS. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 472–482, 1986.

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

[29] F. Matthes, G. Schroder, and J. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.

[30] R. Morrison, R. C. H. Connor, G. N. C. Kirby, D. S. Munro, M. P. Atkinson, Q. I. Cutts, A. L. Brown, and A. Dearle. The Napier88 persistent programming language and environment. In M. P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.

[31] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL)*, pages 221–232, 2005.

[32] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[33] T. Rus and E. Van Wyk. A formal approach to parallelizing compilers. In *Proc. of the SIAM Conf. on Parallel Processing for Scientific Computation*, March 14 1997.

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

[35] J. Schmidt, F. Matthes, and P. Valduriez. Building persistent application systems in fully integrated data environments: Modularization, abstraction and interoperability. In *Proceedings of Euro-Arch'93 Congress*. Springer Verlag, Oct. 1993.

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

[37] R. Software. Whitepaper on the UML and Data Modeling, 2000.

[38] R. Vitenberg, K. Kvilekval, and A. K. Singh. Increasing concurrency in databases using program analysis. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, pages 341–363, 2004.

[39] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[40] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.

[41] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying in Java. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, 2006.