

Query Anomalies

Mandana Vaziri Vladimir Gapeyev Stephen Fink Frank Tip Julian Dolby

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

{mvaziri,sjfink,ftip,dolby}@us.ibm.com, vgapeyev@acm.org

1. Introduction

Several languages including C_{ω} [1] and Linq [2] support first-class query constructs. These constructs are useful for navigating the heap at a higher level of abstraction, by selecting objects by way of relational expressions rather than by dereferencing pointers. However, there is some difficulty in defining the semantics of relational operators in the face of mutable state. The meaning of such operators is linked to the notion of equality. In the absence of mutation, two tuples (instances of a relation) are equal if they have all the same elements, that is, they are structurally equivalent. On the other hand, when tuples are allowed to be mutated, it is not clear what equality means. One possible meaning is simply object equality. The discrepancy between object equality and structural equivalence gives rise to a number of undesirable behaviors that we call *query anomalies*. This position paper illustrates what these are.

2. Query Anomalies

2.1 Examples

```
class Student {      class Course {      class Attends {
  int ssn;           int code;           Student student;
  String name;      String name;       Course course;
}                   }                   }

Set<Attends> attendsRel;
```

Figure 1. A simple student-course relation

In Java, a simple way of implementing tuples and relations (i.e., sets of tuples) relies on the use of classes and collections. In the example of Figure 1, `attendsRel` is a relation, a collection of `(Student, Course)` pairs. As written, it is possible to have two objects of type `Attends` in `attendsRel` that contain the same data, i.e. students with the same `ssn` and name, and courses with the same code and title. This is undesirable, as clearly two students with the same `ssn` are intended to be equal, because `ssn` is assumed to be a unique identification code for students. If the collection behaves like a bag, then it is harder to answer questions such as how many students there are taking some course. To remedy this, in Java, we can define `equals()` and `hashCode()` methods for each class that take into account all its fields, as shown in Figure 2.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]...\$5.00.

```
class Student {
  int ssn;
  String name;

  public boolean equals(Object o){
    if (o != null &&
        this.getClass().equals(o.getClass())) {
      Student other = (Student) o;
      return this.ssn == other.ssn
             && this.name.equals(other.name);
    }
    return false;
  }
  public int hashCode() { ... }
}

class Course {
  int code;
  String title;

  public boolean equals(Object o){
    if (o != null
        && this.getClass().equals(o.getClass())) {
      Course other = (Course) o;
      return this.code == other.code
             && this.title.equals(other.title);
    }
    return false;
  }
  public int hashCode() { ... }
}

class Attends {
  Student student;
  Course course;

  public boolean equals(Object o){
    if (o != null
        && this.getClass().equals(o.getClass())) {
      Attends other = (Attends) o;
      return this.student.equals(other.student)
             && this.course.equals(other.course);
    }
    return false;
  }
  public int hashCode() { ... }
}
```

Figure 2. Adding `equals()` and `hashCode()` methods

```

Student s1 = new Student(1, 'Anne');
Course c1 = new Course(1, 'Algebra');
Attends a1 = new Attends(s1,c1);
attendsRel.add(a1);
if (attendsRel.contains(a1)) //returns true
...
s1.name = 'Anne-Marie';
if (attendsRel.contains(a1)) //returns false!
...

```

Figure 3. Example of a tuple that disappears from a relation

```

Student s1 = new Student(1, 'Alice');
Student s2 = new Student(1, 'Bob');
Course c1 = new Course(2, 'Algebra');
Attends a1 = new Attends(s1, c1);
Attends a2 = new Attends(s2, c1);
attendsRel.add(a1);
if (attendsRel.contains(a2)) //returns true!
...

```

Figure 4. Example of a tuple that appears in a relation

Now consider the fragment of code in Figure 3. This code demonstrates undesirable behavior since an entire tuple seems to have disappeared from the relation, without ever having been removed, because we changed one field of one of its elements. Clearly, if student Anne changes her name to Anne-Marie, we don't want all her data to disappear from all the relations relevant to her. We consider this to be a query anomaly, since the second containment test returns an unexpected result. One way to remedy this is to carefully choose the fields that are used in equality comparisons and hashCode computations. For example, we would not have the strange behavior above if class Student only uses ssn in its equals() and hashCode() methods.

However, a different anomaly is still possible as shown in Figure 4. The tuple (s2, c1) appears to be in the relation attendsRel even though it was never added. If the programmer prints the content of attendsRel, (s2, c1) will indeed not be listed. However the test returns true because a1 and a2 are equal despite containing different data. This behavior is clearly error-prone.

To illustrate anomalies involving relational operators such as join, consider the code example in Linq shown in Figure 5. The output of Test shows that the result of the join operation is empty, even though course c1 and c11 have the same data and ought to be equal tuples. These two objects are not equal because they have different object identifiers (OIDs).

To remedy this anomaly, the programmer may add Equals() and GetHashCode() methods for Course as shown in Figure 6. Consider now the fragment of code in Figure 7. When the title of the course is changed, it disappears from the table. To prevent this behavior, the programmer may want to define Equals() and GetHashCode() to use only the course code. This will ensure that 2 courses are equal if they have the same code. Then the title may be changed without affecting membership in collections such as hashtable and hashsets (Figure 8).

Consider now the code of Figure 9. The output is Student:Alice Faculty:Bob Error:True In this example, the join occurs on two courses (c1 and c11) that are equal but do not have equal elements (different titles). Two

```

namespace SampleQueries {
public class Faculty {
public int ssn;
public String name;

public Faculty(int ssn, String name){
this.ssn = ssn;
this.name = name;
}
}

public class Student {
public int ssn;
public String name;

public Student(int ssn, String name){
this.ssn = ssn;
this.name = name;
}
}

public class Course {
public int code;
public String title;

public Course(int code, String title){
this.code = code;
this.title = title;
}
}

public class Attends {
public Student student;
public Course course;

public Attends(Student student, Course course){
this.student = student;
this.course = course;
}
}

public class Teaches {
public Faculty faculty;
public Course course;

public Teaches(Faculty faculty, Course course){
this.faculty = faculty;
this.course = course;
}
}

public void Test(){
Student s1 = new Student(1, "Alice");
Faculty f1 = new Faculty(2, "Bob");
Course c1 = new Course(1, "Algebra");
Course c11 = new Course(1, "Algebra");
Attends a1 = new Attends(s1,c1);
List<Attends> attends = new List<Attends>();
attends.Add(a1);
Teaches t1 = new Teaches(f1, c11);
List<Teaches> teaches = new List<Teaches>();
teaches.Add(t1);
var q =
from a in attends
join t in teaches on a.course equals t.course
select new { Student = a.student,
Faculty = t.faculty};
foreach (var v in q){
Console.WriteLine("Student:" + v.Student.name
+ " Faculty:" + v.Faculty.name);
}
}
}

```

Figure 5. Linq Example

```

public class Course {
    public int code;
    public String title;

    ...
    public override bool Equals(Object o) {
        if (o != null
            && this.GetType().Equals(o.GetType())) {
            Course other = (Course) o;
            return this.code == other.code
                && this.title.Equals(other.title);
        }
        return false;
    }

    public override int GetHashCode() {
        return 6101 * code + 6113
            * ((title == null) ? 1 : title.GetHashCode());
    }
}

```

Figure 6. Adding Equals() and GetHashCode() methods

```

public void Test(){
    Course c11 = new Course(1, "Algebra");
    Hashtable table = new Hashtable();
    table.Add(c11, "stats");

    if (table.Contains(c11)) //Returns true
        ...

    c11.title = "Algebra I";

    if (table.Contains(c11)) //Returns false!
        ...
}

```

Figure 7. Tuple disappearing

```

public class Course {
    public int code;
    public String title;

    ...
    public override bool Equals(Object o) {
        if (o != null
            && this.GetType().Equals(o.GetType())) {
            Course other = (Course) o;
            return this.code == other.code;
        }
        return false;
    }

    public override int GetHashCode() {
        return 6101 * code;
    }
}

```

Figure 8. Redefining Equals() and GetHashCode()

```

public void Test(){
    Student s1 = new Student(1, "Alice");
    Faculty f1 = new Faculty(2, "Bob");
    Course c1 = new Course(1, "Algebra");
    Course c11 = new Course(1, "Algebra");
    Attends a1 = new Attends(s1,c1);
    List<Attends> attends = new List<Attends>();
    attends.Add(a1);
    Teaches t1 = new Teaches(f1, c11);
    List<Teaches> teaches = new List<Teaches>();
    teaches.Add(t1);

    c11.title = "Algebra I";

    var q =
        from a in attends
        join t in teaches on a.course equals t.course
        select new { Student = a.student,
                    Faculty = t.faculty ,
                    Error = (! a.course.title.Equals(t.course.title));}

    foreach (var v in q){
        Console.WriteLine("Student:" + v.Student.name +
            " Faculty:" + v.Faculty.name
            + " Error:" + v.Error);
    }
}

```

Figure 9. Joining on tuples that are not structurally equivalent

tuples may be equal, but this doesn't mean that all their elements are equal. The programmer needs to be aware of the implementation of Equals() and GetHashCode() in order to know what fields of the 2 courses can be assumed to be equal. This involves non-local reasoning and is error-prone.

To ensure that the membership of tuples in relations is unchanged when mutating certain elements, the programmer must define Equals() and GetHashCode() methods on a subset of the fields. This implies that equality is no longer the same as structural equivalence. However, when a tuple is contained in a relation, or when elements of two tuples match when joining them, the programmer expects structural equivalence. But this may not always be the case. The difference between object equality and structural equality gives rise to these anomalies.

2.2 Anomalies

In this section, we present query anomalies more formally. A *tuple* t is a sequence of *elements*. An element is either a value or another tuple. In an object-oriented language, an object can be viewed as a tuple, whose elements are its fields. We use the notation t_i to denote the i^{th} element of t . We write \emptyset to denote the empty tuple that has no elements. The function *empty*(t) tests if t is the empty tuple. The functions *first*(t) and *last*(t) give the first and last elements of t respectively. Let *update*(t) be a procedure that changes some elements of t (we do not further specify the exact behavior of *update*(t)).

We define two different notions of equality on tuples. The first, denoted $=_o$, models object equality. In practice, it is either defined by object identifier (OID) comparisons, or explicitly given with equals() and hashCode() methods. The second notion of equality, denoted \equiv , corresponds to structural equivalence and is defined as follows.

Definition Structural equivalence. $t \equiv t'$ if and only if t and t' have

the same number of elements, and for each i :

$$\begin{cases} t_i \text{ is the same as } t'_i & \text{if } t_i \text{ and } t'_i \text{ are both values,} \\ t_i \equiv t'_i & \text{otherwise} \end{cases}$$

A relation s is a set of tuples. Let $member(t, s)$ be a function to test the membership of a tuple t in a set s , defined as follows:

$$member(t, s) \text{ if and only if } \exists t' \in s \mid t = t'$$

We also define two procedures $add(t, s)$ and $remove(t, s)$ defined with the following Hoare triples ($\{\text{precondition}\}$ command $\{\text{postcondition}\}$).

$$\{\text{True}\} add(t, s) \{member(t, s)\}$$

$$\{\text{True}\} remove(t, s) \{\neg member(t, s)\}$$

The function $join(t, t')$ takes two tuples t and t' , and creates a new tuple that is the result of performing a join on the first and last elements. More formally, let t_n be $last(t)$ and t'_m be $last(t')$:

$$join(t, t') = \begin{cases} (t_0, \dots, t_{n-1}, t'_1, \dots, t'_m) & \text{if } last(t) =_o first(t') \\ \emptyset & \text{otherwise} \end{cases}$$

The following series of Hoare triples are query anomalies.

Query Anomalies:

1. $\{member(t, s)\} update(t) \{\neg member(t, s)\}$
2. $\{t \neq t' \wedge t =_o t' \wedge \neg member(t', s)\} add(t, s) \{member(t', s)\}$
3. $\{t \neq t' \wedge t =_o t' \wedge member(t', s)\} remove(t, s) \{\neg member(t', s)\}$
4. $\{t \equiv t' \wedge t \neq_o t' \wedge member(t', s)\} \{\neg member(t, s)\}$
5. $\{last(t) \equiv first(t') \wedge last(t) \neq_o first(t')\} \{\emptyset\}$
6. $\{last(t) \neq first(t') \wedge last(t) =_o first(t')\} \{\emptyset\}$

Anomaly 1 represents a behavior where t is initially a member of set s , and is no longer a member as a result of an update. Clearly, changing the content of a tuple should not affect its membership in sets containing it. Anomalies 2 through 6 show undesirable behaviors that arise when the notions of object equality and structural equality do not coincide. Anomalies 2 and 3 deal with a tuple appearing or disappearing in a set when an unrelated tuple is added or removed, respectively. In anomaly 4, t and t' are structurally identical. One could mistakenly expect that if t belongs to a set, then so does t' . But this is not necessarily the case because $t \neq_o t'$. Finally, anomalies 5 and 6 consider unexpected behavior involving joins. In anomaly 5, t and t' have last and first elements that match structurally. One may expect the resulting join not to be the empty tuple, but this will be the case because the last and first elements are not equal. Conversely, in anomaly 6, t and t' have last and first elements that are not structurally equivalent, so one may expect the join operation to result in the empty tuple. But this is not the case because $last(t) =_o first(t')$.

The examples presented previously illustrate some of these anomalies. Figure 3 shows an instance of anomaly 1. Once $s1$ is mutated, the hash-code for $a1$ may change, and it may thereafter belong to the wrong bucket in hashset `attendsRel`. Since the hashset contains only elements placed in the correct bucket (these are the ones for which `contains()` returns true), $a1$ has effectively disappeared from the relation. In our formal model, $a1$ no longer belongs to the set represented by the hashset. Figure 4 is an illustration of anomaly 2. Figures 5 and 9 illustrate anomalies 5 and 6.

2.3 Desired Properties

The following properties in a language design for tuples and relations would address query anomalies:

Desired Properties:

- A. $t =_o t' \Leftrightarrow t \equiv t'$
- B. $\{t =_o t'\} update(t) \{t =_o t'\}$

Property A states that the notions of object equality and structural equality are the same for all tuples. This property guarantees that anomalies 2 through 6 do not occur, because it is not possible for two tuples to be structurally equivalent but not equal or vice versa.

Property B states that the *identity* of a tuple does not change when it is updated. That is, if two tuples t and t' are equal, an update to t will not change this equality. Property B guarantees that anomaly 1 cannot occur by the following reasoning. Let t and s be such that $member(t, s)$. By the definition of membership, this means that there exists a tuple t' in s such that $t =_o t'$. Given property B, we know that after an update to t , it is still the case that $t =_o t'$. Therefore we have that $member(t, s)$ after the update as well, which means that anomaly 1 cannot occur.

3. Conclusion

In the future, we plan to investigate language constructs for relations and querying that address these anomalies. In particular, we believe relation types [3] might provide a starting point.

Acknowledgements

We thank the referees for their insightful comments.

References

- [1] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in C_ω . In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of Lecture Notes in Computer Science, pages 287311. Springer-Verlag, 2005.
- [2] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.
- [3] M. Vaziri, F. Tip, S. Fink, and J. Dolby. “Declarative Object Identity using Relation Types”, In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'07)*, July 2007.