Bi-CQ: A Bidirectional Code Query Language

Dongxi Liu, Yingfei Xiong, Zhenjiang Hu, and Masato Takeichi

Department of Mathematical Informatics
University of Tokyo
{liu, hu, takeichi}@mist.i.u-tokyo.ac.jp,
xiong.yingfei@gmail.com

1 Introduction

Bi-CQ is a bidirectional language for querying and updating Java source code, being developed in PSD Lab, University of Tokyo. Each query in Bi-CQ can be executed in two directions. The forward execution of a Bi-CQ query, just like querying in the existing code query languages [1], returns the expected program fragments from the source code. The difference is that with Bi-CQ users can update the returned program fragments, and the updates can be put back into the source code automatically after executing the same query backward. An update indicates a piece of code to run before, after or around a program fragment in the query result.

2 An Example

Consider the Java source code in Figure 1(a). For some reason, we are interested in querying all method invocations `a1(n)` in this source. This kind of code query can be implemented by using Bi-CQ. The Bi-CQ code in Figure 1(b) returns the method invocation in method `a2` after forward execution. For this query result, we are not satisfied with only taking a look at them, and rather we want to change the source to print the integer argument used in this call. Fortunately, Bi-CQ still can help us. With the update in Figure 1(c), the same query can be executed backward, and after this the source code is changed to reflect this update. The updated source code is given in Figure 2. This example can be found at [3].

```
class A{
  int a1(int n){return n++;}
  int a2(int n){return a1(n)+1;}
}
```
(a)

```
xclass; xmethod;
xmap(xbody;xretexpr;
     xif (xisnode InfixExpression)
        xleftexpr xhide )
```
(b)

```
before update(int x){
  System.out.println("The argument is "+Integer.toString(x));
}
```
(c)

Fig. 1. Example: (a) Java Source Code, (b) Bi-CQ Query and (c) Update

```
class A {
  int a1(int n){return n++;}        int a2(int n){return call_a1_64(null,n)+1;}
  int call_a1_64(A rev, int n){
    int x=n;     {System.out.println("The argument is " + Integer.toString(x));}    return a1(n); }
}
```

Fig. 2. Example: Updated Source

3 Design of Bi-CQ

The design of Bi-CQ is based on a bidirectional language defined in [2]. Some constructs in [2] are specific for constructing and deconstructing XML data, while others, such as let-binding and function-call, are more generic. Bi-CQ inherits all those generic constructs, enhanced with new constructs for processing Java source code.

Bi-CQ has some new constructs to implement navigation through program structures, taking the same role as those to deconstruct XML data. However, for these constructs in Bi-CQ, it is not always acceptable to put modifications on query results directly back into the corresponding points in the source, and rather their backward behaviors must guarantee the updated source still has correct syntax and semantics if the original one has. For the example above, it is not correct in syntax if the backward execution puts the print statement directly before the expression `a1(n)` in the return statement.

The update code in the source may be expected to execute under some dynamic conditions. For the example above, users may only want to print the argument of `a1(n)` if it is called during the execution of another method. Bi-CQ deals with this case by providing users constructs to express assumptions on the source when writing queries. And during their backward executions, these assumptions will be asserted by inserting appropriate code in the source. There are some examples about these constructs at [3].

4 Applications of Bi-CQ

One intended application of Bi-CQ is to define the semantics of AspectJ language, motivated by [4]. In our approach, pointcuts are translated into Bi-CQ queries, and advised code into updates. After backward execution of queries, the advised code and the code for supporting dynamic pointcuts are automatically instrumented into the points in the source specified by static pointcuts. The advantage of our approach is that it can help understand how interactions occur among static pointcuts, dynamic pointcuts, advised code and mainline programs in AspectJ.

References

1. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: scalable source code queries with datalog. In *ECOOP*, 2006.
2. Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of XQuery. In *PEPM*, 2007.
3. Bidirectional Code Query. http://www.ipl.t.u-tokyo.ac.jp/~liu/Bi-CQ.html.
4. Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in aspectj. In *POPL*, 2007.