Amos II Java Interfaces

Daniel Elin and Tore Risch

Uppsala Database Laboratory

Department of Information Science

Uppsala University

Sweden

Tore.Risch@dis.uu.se

2000-08-25

Latest modification 2006-02-17

This report describes the external interfaces between Amos II and the programming language Java. There are also interfaces between Amos II and the programming languages C and Lisp, documented separately. There are mainly two ways to interface Amos II with Java programs: Either a Java program calls Amos II through the *callin* interface, or foreign AMOSQL functions are defined by Java methods through the *callout* interface. The combination is also possible whereforeign functions in Java call Amos II back through the callin interface.

1.   Introduction

This document assumes that you have read the overview of Amos II [5].

There are external interfaces between Amos II and the programming languages ANSII C, Lisp, and Java. This document describes Java interfaces, while the the C and Lisp interfaces are documented separately in [3]. The Java interface is the most convenient way to write Amos II applications and to do simple extensions. The C and Lisp interfaces must be used for more advanced system extensions and time critical applications.

Some advantages with the Java interfaces compared to the C interfaces are:

1. Programming errors in the Java code cannot cause Amos II to crash. By contrast, programming errors in C can cause very nasty system errors. The Lisp interfaces are much safer than C, but not as protected (and limited) as the Java interfaces.

2. Memory is deallocated automatically in Java through its automatic garbage collector. In C/C++ memory deallocation is manual and very error prone, while Lisp also has automatic garbage collection.

3. Java has extensive libraries for user interfaces, web accesses, etc. which can easily be interfaced from Amos II through its Java interfaces.

The disadvantages with the Java interfaces are that they are slower and more limited than the other ones.

There are two main kinds of external interfaces, the *callin* and the *callout* interfaces:

*   With the *callin* interface a program in Java calls Amos II. The callin interface is similar to the call level interfaces for relational databases, such as ODBC, JDBC, ORACLE call-level interface, etc.

*   With the *callout* interface Amos II functions call methods written in Java. Each such *foreign AMOS function* is implemented by one or several Java methods. The callout interface has similarities with 'data blades' in Object-Relational databases [7]. The foreign functions in Amos II are *multi-directional* which allow them to have separately defined inverses and to be indexed [2]. The system furthermore allows the callin interface to be used also in foreign functions, which gives great flexibility and allows Java methods to be used as a form of *stored procedures*.

With the callin interface there are two ways to call Amos II from Java:

*   In the *embedded query* interface a Java method is provided that passes strings containing AMOSQL statements to Amos II for dynamic evaluation. Methods are also provided to access the results of the dynamically evaluated AMOSQL statement. The embedded query interface is relatively slow since the AMOSQL statements have to be parsed and compiled at run time.

*   In the *fast-path* interface predefined Amos II functions are called as Java methods, without the overhead of dynamically parsing and executing AMOSQL statements. The fast-path is significantly faster than the embedded query interface. It is therefore recommended to always make Amos II derived functions and stored procedures for the various Amos II operations performed by the application and then use the fast-path interface to invoke them directly [1].

There are also two choices of how the connection between application programs and Amos II is handled:

1. Amos II can be linked directly with a Java application program. This is called the *tight connection* where Amos II is an *embedded database* in the application. It provides for the fastest possible interface between an application and Amos II since both the application and Amos II run in the same address space. The disadvantages with the tight connection is that Amos II and the application must run on the same computer. Another disadvantage is that only a single Java application can be linked to Amos II.

2. The Java application can be run as a client to an Amos II server. This is called the *client-server connection*. With the client-server connection several applications can access the same Amos II concurrently. The Java applications and the Amos II server run as different programs. However, the overhead of calling Amos II from another program can be several hundred times slower than the tight connection. Currently the client-server connection also requires

Amos II kernel code (in C) to be called from Java, and Amos II Java clients are thus not applets.

The *callout* interface is implemented through a mechanism called the *multi-directional foreign function* interface. For example, external databases can be accessed through Java and made fully available in Amos II. Programs called through the callout interface always run in the same address space as the Amos II server. With the Java callout interface it is easy to extend Amos II with simple foreign Amos II functions written in Java.

This document is organized as follows:

- Sec. 2. documents the *callin* interface to Amos II from Java using either the embedded query or the fast-path interface.

- Sec. 3.describes the *callout* interface where Amos II functions can be defined in Java.

- Sec. 4. describes how to customize the query optimizer with *cost hints* for the callout functions.

2.   The Callin Interface

The following things should be observed when calling Amos II from Java:

- There exists both an *embedded query* interface (Sec. 2.3) and a *fast-path* interface (Sec. 2.6). The fast-path interface is up two orders of magnitude faster.

- There is a *tight connection* between an application and Amos II where the Amos II is linked to a Java application program. Two DLLs (*amos.dll* and *JavaAMOS.dll*) and a JAR file (*javaamos.jar*) are provided for this.

- In order to call Amos II from Java a *driver program* (Sec. 2.1) is needed. It is a Java main program that sets up the Amos II environment and then calls the Amos II kernel. The driver program JavaAMOS (in *javaamos.jar*) is a very simple driver program that simply calls the AMOSQL top loop [1]. If you use only the callout interface you can use this driver.

- Data is passed between the application and Amos II using some special Java classes defined in *javaamos.jar*.

The system is delivered with sample Java programs that demonstrates the callin (*AmosDemo.java*), and the callout (*Foreign.java*) interfaces, respectively.

2.1   The driver program

This is an example of a driver program; it shows the complete code of the system driver *JavaAMOS*:

```
import callin.*; // The Amos II callin interface
/**
 The standard JavaAMOS driver program.
 This is the typical format of all JavaAMOS drivers
 started from the command line.
 */
public class JavaAMOS {

 public static void main(String argv[]) throws AmosException
 {
   Connection.initializeAmos(argv);  // Can only be called once to initialize the Amos II kernel
   Connection theConnection = new Connection("");
   theConnection.amosTopLoop("JavaAMOS"); // Enters the AMOSQL top-loop
 }
}
```

Amos II must first be initialized with the method

> Connection.initializeAmos(String imageName) throws AmosException;

The parameters specified on the command line are passed to the Amos II kernel through argv for interpretation. initializeAmos knows how to handle a user specified name of a database image on the command line.

If the initialization failed the system will throw the exception AmosException; it can be caught using Java's exception handling mechanisms.

In order to access an Amos II database an object of class Connection must be allocated first. The constructor takes the name of the Amos II database server to which the connection is to be established as parameter. If the empty string in specified, as in the example, it indicates a *tight* connection to an embedded Amos II database. If a *client-server* connection is to be established this constructor takes the name of the Amos II server to connect to as parameter.

To enter the interactive AMOSQL top loop from C, call the method

> void amosTopLoop(String prompt);

amosTopLoop returns if the AMOSQL command 'exit;' is executed. By contrast, the AMOSQL command 'quit;' terminates the program.

2.2   Connections

The class Connection represents connections to Amos II databases. The constructor is:

> Connection(String dbName) throws AmosException;

If dbName is the empty string it represents a tight connection to the embedded database; otherwise dbName must be the name of an Amos II mediator peer known to the Amos II nameserver [5] running on the same computer as the application. If you with to connect to an Amos II peer managed by a nameserver running on the computer with IP address nameServerHost use this constructor instead:

> Connection(String dbName, String nameServerHost) throws AmosException;

A connection can be terminated when no longer used with the Connection method:

> void disconnect() throws AmosException;

Connections are automatically terminated when deallocated by Java.

2.3   The Embedded Query Interface

In the *embedded query* interface, strings with AMOSQL statements can be sent to the Amos II kernel for execution by calling the Connection method:

> Scan execute(String query) throws AmosException;

For example

> theScan = theConnection.execute("select name(t) from type t;");

The query is a C string with an AMOSQL statement ended with';'. The result of the query is a *Scan* object associated with the connection. A scan is a stream of *tuples* (rows) which can be iterated through with special interface functions as explained below.

If the number of tuples returned in the scan is large it is sometimes desired to limit the number of tuples to return (i.e. limiting the size of the scan). A second variant of execute is used for this where the parameter stopAfter specifies how many tuples are maximally returned in the scan:

> Scan execute(String query, int stopAfter) throws AmosException;

2.3.1 Scans

In the embedded query interface, but also when Amos II functions are called through the *fast-path* interface, the results are returned as *scans*.

Use the following method to iterate through all tuples in a scan s:

```
...
theScan = theConnection.execute("select name(t) from type t;");
while (!theScan.eos()) // While there are more rows in the scan
{
        Tuple row; // Will hold each result tuple in scan
        String str; // Will hold each type name in each result tuple

        row = theScan.getRow();    // Get current row in the scan
        str = row.getStringElem(0);   // Get 1st element (enumerated 0 and up) in row as a string
        // Here you have access to each type name
         theScan.nextRow();    // Advance the scan forward
}
```

The Scan method eos tests if there are any more rows in a scan:

```
boolean eos();
```

The Scan method getRow picks up the current tuple (initially 1st in the scan) from the scan:

```
void getRow(Tuple preAlloc) throws AmosException;
```

The Scan method nextRow advances the scan forward, i.e. sets the current tuple to the next tuple:

```
void nextRow() throws AmosException;
```

The scan is automatically *closed* by the system when all tuples have been iterated through or it is deallocated by the garbage collector. In case you for some reason need to close the scan prematurely you can close it explicitly by calling the Scan method closeScan:

```
void closeScan() throws AmosException;
```

2.3.2 Single tuple results

Often the result of a query is a single tuple. In those cases there is no need to iterate through the scan (or close it) and the value can be accessed by this method:

```
String str;
...
str = theConnection.execute("concat('a','b');",1).getRow.getStringElem(0);
```

In this case the result string is the first element (position 0) of the first row returned from the AMOSQL statement.

2.4 Object Proxies

Object Proxies are Java objects representing a corresponding referenced Amos II database objects. An object proxy represents a reference to an Amos II database object. Object proxies can reference any kind of data stored in Amos II databases, including number, strings, surrogate objects, arrays, and other internal Amos II data structures. Object proxies are represented using the class Oid.

The name of the Amos II datatype of an object proxy can be retrieved with the method:

```
String getTypename() throws AmosException;
```

The Amos II datatype object of an object proxy can be retrieved with the method:

```
Oid getType() throws AmosException;
```

The connection to the Amos II system owning an object proxy can be retrieved with:

```
Connection getConnection() throws AmosExcetion;
```

2.5   Tuples

Tuple is a commonly used datatype in the Amos II interface. Tuple objects are represented as object of class Tuple. A tuple represents an ordered collection of Amos II database objects. Tuples are used for many purposes in the interfaces:

- Tuples are used when iterating through scans from results of Amos II commands and function calls, as illustrated in Sec. 2.3.1.
- Tuples are used for representing argument lists in Amos II function calls (Sec. 2.6).
- Tuples represent sequences (Sec. 2.5.5) of data values.
- Tuples are used in the *callout* interface for interfacing argument lists of foreign AMOS functions defined in Java.

There are a number of system functions for moving data between tuples and other types of Java objects.

The Tuple method getArity returns the number of elements in the tuple, its arity.

```
int getArity() throws AmosException;
```

To allocate a new tuple with the a given arity and use the Tuple constructor:

```
Tuple(int arity);
```

The elements of the argument list should then be assigned with some tuple update function as explained below.

The elements of a tuple are enumerated starting at 0 and can be accessed through a number of tuple access functions specific for each element class, as described next.

2.5.1   String elements

To copy data from an element in a tuple into a Java string use the Tuple method getStringElem:

```
String getStringElem(int pos) throws AmosException;
```

It copies element number pos of the tuple into a Java string. An error occurs if the element is not a string.

To store a string str in element pos of a tuple use the overloaded Tuple method:

```
void setElem(int pos, String str) throws AmosException;
```

For example, the following code creates a tuple containing a single string:

```
Tuple tp = new Tuple(1);
tp.setElem(0,"HELLO WORLD");
```

2.5.2   Integer elements

To get an integer stored in position pos of a tuple use the Tuple method getIntElem:

```
int getIntElem(int pos) throws AmosException;
```

An error is generated if there is no integer in the specified position of the tuple.

To store an integer in element pos of the tuple use the overloaded Tuple method:

```
void setElem(int pos, int integer) throws AmosException;
```

2.5.3   Floating point elements

To get a double precision floating point number stored in position pos of a tuple use the Tuple method getDoubleElem:

```
double getDoubleElem(int pos) throws AmosException;
```

An error is generated if there is no a real number in the specified position of the tuple.

To store a floating point number in element pos of a tuple use the overloaded Tuple method setElem:

```
void setElem(int pos, double dbl) throws AmosException;
```

2.5.4   Object elements

To get a proxy for the object stored in position pos of a tuple use the Tuple method getOidElem:

```
Oid getOidElem(int pos) throws AmosException;
```

It returns an Amos II *object proxy* which represents any kind of data structure supported by the Amos II storage manager, as explained in Sec. 2.4.

To store an object handle in element pos of a tuple use the overloaded Tuple method setElem:

```
void setElem(int pos, Oid obj) throws AmosException;
```

2.5.5   Sequences

Sequences are special Amos II objects containing several other objects with significant order represented as objects of type *vector* in Amos II databases. Sequences are also represented as as objects of Tuple in Java.

To get a sequence stored in position pos of a tuple use the Tuple method getSeqElem:

```
Tuple getSeqElem(int pos) throws AmosException;
```

An error is generated if there is no sequence in the specified position of the tuple. The usual tuple access methods can then be used for accessing the elements of the sequence.

To store a copy of the tuple tpl as a sequence in element pos of a tuple use the overloaded Tuple method setElem:

```
void setElem(int pos, Tuple tpl) throws AmosException;
```

2.5.6   Element data type tests

The data type of an element in position pos of a tuple can be tested with these Tuple methods:

```
boolean isString(int pos) throws AmosException;
boolean isInteger(int pos) throws AmosException;
boolean isDouble(int pos) throws AmosException;
boolean isObject(int pos) throws AmosException;
boolean isTuple(int pos) throws AmosException;
```

2.6   The Fast-Path Interface

The *fast-path* interface permits Amos II functions to be called from Java. Primitives are provided for building argument

lists of Java data to be passed as parameters to a Amos II function call. The result of a fast-path function call is always as scan which is treated in the same way as the result of an embedded query call (Sec. 2.3).

The following example shows how to call the Amos II function charstring.lower->charstring (which lower cases a string) from Java:

```
Tuple argl; // The actual argument list
String res; // The result string

argl = new Tuple(1);  // There is one argument
argl.setElem(0, "HELLO WORLD"); // the string argument is inserterd into the argument list
res = theConnection.callFunction("charstring.lower->charstring",argl).getRow().getStringElem(0);
// res will now contain the string 'hello world'
```

Argument lists represent arguments to be passed to fast-path AMOS function calls. They are represented as Java objects of type Tuple. Before an Amos II function is called an argument list must be allocated with the correct arity of the function to call by using the Tuple constructor:

```
Tuple(int arity)
```

The elements of the argument list should then be assigned with some tuple update function (Sec. 2.5). The first element in the tuple (position 0) is the first argument, etc.

When all arguments in the argument list are assigned the function named fnName can be applied on the argument list fnArgs with the fnName method callFunction:

```
Scan callFunction(String fnName, Tuple fnArgs) throws AmosException;
Scan callFunction(String fnName, Tuple fnArgs, int stopAfter) throws AmosException;
```

The second variant is to limit the scan size as for fnName. The result of the function call is a scan.

Notice here that the Java interface caches the association between the name of an Amos II function, fnName, and the corresponding *function proxy* representing the database function with that name. This has two consequences:

1. The first call to callFunction accesses the database to get the function proxy and the subsequent calls will then be very fast.

2. callFunction will not notice if the name of the function has been changed between two calls, and thus the function proxy used by the first call will be returned even if the name of the function has changed in the meantime.

If required, the function proxy cache can be cleared by calling the Connection method clearFunctionCache:

```
void clearFunctionCache();
```

To explicitly get the function proxy, given a function name, use the Connection method getFunction:

```
Oid getFunction(String fnName) throws AmosException;
```

callFunction is overloaded to also accept function proxies instead of function names:

```
Scan callFunction(Oid fnObject, Tuple fnArgs) throws AmosException;
Scan callFunction(Oid fnObject, Tuple fnArgs, int stopAfter) throws AmosException;
```

2.7   Creating and Deleting Objects

This subsection describes how to create new Amos II database objects from Java. A new such database object of an Amos II type named TypeName is created with the Connection method createObject:

```
Oid createObject(String TypeName) throws AmosException;
```

As for function proxies the system internally maintains object proxies for types too, called *type proxies*. When createObject is passed the name of a type it first looks up the database to find the database object representing that type and then create the corresponding type proxy in Java. As for function proxies the type proxies are cached in the Java mem-

ory. In the very unlikely situation that some cached type proxy has become invalid[1] the type cache can be cleared with the Connection method clearTypeCache:

    void clearTypeCache();

You can also explicitly obtain the type proxy with the Connection method getType:

    Oid getType(String typeName) throws AmosException;

It returns the type proxy for the type named typeName.

createObject is overloaded to also accept type proxies:

    Oid createObject(Oid type) throws AmosException;

where type is the type proxy for the new object. createObject creates a new database object and returns its object proxy.

To delete a database object theObject use the Connection method deleteObject:

    void deleteObject(Oid theObject) throws AmosException;

2.8   Updating Amos II functions

Stored Amos II functions and some derived functions can be updated, i.e. assigned a new value for a given argument combination. The following is an example of how to create an object of type person and then update the function name for the new object:

```
Connection theConnection;
Oid nw;
...
nw = theConnection.createObject("person");
Tuple argl = new Tuple(1);
Tuple resl = new Tuple(1);
argl.setElem(0,nw);
resl.setElem(0,"Tore");
theConnection.addFunction("name",argl,resl);
```

The Connection method setFunction updates an Amos II function:

    void setFunction(String fnName, Tuple argList, Tuple resList) throws AmosException;
    void setFunction(Oid fn, Tuple argList, Tuple resList) throws AmosException;

The first variant uses the function proxy cache. The second variant requires that a function proxy has been previously retrieved, e.g. with getFunction.

The Connection method addFunction adds a new value to a bag valued Amos II function:

    void addFunction(String fnName, Tuple argList, Tuple resList) throws AmosException;
    void addFunction(Oid fn, Tuple argList, Tuple resList) throws AmosException;

Notice that addFunction and setFunction have the same effect when no previous function value exists. However, since addFunction is faster than setFunction, it is recommended to use addFunction when setting new object properties, as in the example above.

The Connection method remFunction removes a value from a bag valued Amos II function:

    void remFunction(String fnName, Tuple argList, Tuple resList) throws AmosException;
    void remFunction(Oid fn, Tuple argList, Tuple resList) throws AmosException;

1.This can only happen if the type is first deleted and then another type with the same name is created. In that very unlikely case the system will report that the type is not found when using the type proxy.

2.9   Transaction Control

The transaction control primitives only make sense in the embedded database. For client/server calls Amos II every Amos II call is a separate transaction (autocommit), so the transaction primitives will then be treated as dummy operations.

Local transactions are committed with the Connection method:

        void commit() throws AmosException;

Local transactions are rolled back (aborted) with the Connection method:

        void rollback() throws AmosException;

3.   The Callout Interface

This section explains how to implement foreign Amos II functions in Java through the callout interface. The functions can be defined in Java through a special mechanism called the *multi-directional foreign function* interface. Foreign functions used in AMOSQL queries should be side-effect free since the query optimization may rearrange their calling sequence. Foreign functions with side effects (i.e. updating the database) should be used as stored procedures, i.e. not inside queries. There is currently no mechanism in the system to distinguish between foreign function with or without side effects.

Foreign functions in Java are defined as methods of some user defined Java class stored in some external file with the same name as the class. After the class definition has been compiled with the Java compiler it can be dynamically loaded into Amos II by creating a foreign function where the file name and class of the method is specified.

The system will dynamically load the class when the foreign Amos II function is defined. Definitions of foreign functions in Java are saved in the database image. When Amos II is started with an image containing such foreign Java functions it will load its classes before calling the foreign Java function.

We first describe the simplest case, where an Amos II function implemented in Java computes its values given that its arguments are known. We then show how to generalize such *simple foreign Amos II functions* to invertible *multi-directional foreign functions*. The multi-directional foreign function capability allows several access paths to an external data structure, analogous to secondary indexes for relations [2]. Aggregation operators can currently be implemented only in Lisp.

3.1   Simple foreign functions

A simple foreign Amos II function computes its result given some parameters represented as a *tuple*, similar to an ordinary subroutine. The driver program must contain the following code to define foreign functions:

1. Java code to *implement* the function.

2. A *definition* of the foreign function in AMOSQL.

3. An optional *cost hint* (Sec. 4.) to estimate the cost of executing the function.

A foreign function always has two arguments: a *context* and a *parameter tuple*. The context is a data structure managed by the system to pass information between the foreign functions and the rest of the sytem. The parameter tuple is a Tuple object holding both the argument(s) and the result(s) of the function. The foreing function implementation must pick up the argument(s) from the actual argument positions of the parameter tuple and then compute the result(s) and store them in the result positions of the parameter tuple.

For example, the following Java class defines a function sqroot that computes the positive square root of a real number.

It is defined as a method sqrt of the class MyForeignFunctions:

```
import callin.*;
import callout.*;
public class MyForeignFunctions
{
        public void sqroot(CallContext cxt, Tuple tpl) throws AmosException
        {
                double x; // local variable to hold the argument

                x = tpl.getDoubleElem(0);// Pick up the argument
                if (x >= 0.0)
                {
                        tpl.setElem(1, Math.sqrt(x));
                        cxt.emit(tpl);
                }
        }
}
```

The above code must now be placed in a java source file with the same name as the class, *MyForeignFunctions.java*. Notice that the file name *MyForeignFunctions.java* is case sensitive!

Then you can compile the file with

```
javac MyForeignFunctions.java
```

The code is dynamically linked to Amos II with the AMOSQL statement:

```
create function sqroot(real  x)->real r as foreign "JAVA:MyForeignFunctions/sqroot";
```

After this the function sqroot is ready to be used. The string JAVA: indicates that the foreign function is implemented in Java; then follows the name of the foreign class (and file name, *MyForeignFunctions*) followed by a slash and the name of the method of the class implementing the function (sqroot).

3.2   Foreign function Implementation

A foreign function implementation Foreign in Java has the signature:

```
public void Foreign(CallContext cxt, Tuple tpl) throws AmosException;
```

where a CallContext is the context data structure for managing the call and tpl is a *parameter tuple* representing both arguments and results. The first n elements in the parameter tuple represents the actual arguments (n is the function's arity) and the remaining element(s) represents the result(s). Often there is only one result as in the example; boolean valued functions have no result position; functions returning more than one value (width > 1) can have several result positions. The size of the parameter tuple is thus the sum of the arity and the width of the function. In the example abs there is one argument and one result, so the size of tpl is 2 where tpl[0] holds the argument and the computed result is delivered in tpl[1].

The Java method implementing the foreign function should fill in the uncomputed positions of the parameter tuple and emit it with the CallContext method emit:

```
void emit(callin.Tuple tpl);
```

Notice that a result tuple of emit must contain both the values of the input values (bound values) to the implementation and the corresponding output values. The elements of tpl thus represent the combination of the emitted argument values and the corresponding result values.

If emit is not called the result of the foreign function will be empty (nil), as for negative arguments to sqroot.

False values from booean functions are indicated by not calling emit at all (so called negation-by-failure). For example, the following function tests if two text strings have the same letters, independent of their cases:

```
public void eqstring(CallContext cxt, Tuple tpl) throws AmosException
```

```
                    {
                              if(equalsIgnoreCase(tpl.getStringElem[0],tpl.getStringElem[1])) cxt.emit(tpl)
                    }
```

3.2.1   Creating the resolvent for a foreign function

The foreign function must finally be assigned a function resolvent by executing the AMOSQL statement

```
create function <fn>(<argument declaration>) ->
         <result declaration> as foreign 'JAVA:<class file>/<method>';
```

where <fn> is the AMOSQL name of the foreign function, <argument declaration> is the signature of its argument(s), <result declaration> is the signature of its results, <class file> is the name of the class (file) where the function is implemented, and <method> is the name of the method implementing it. The definition can be done from Java by calling the method execute or by an AMOSQL command in the top loop. The definition needs only be done once. It can then be saved in the database image; the system will dynamically load it again whenever the system is initialized with that image.

For example:

```
create function sqroot(real  x)->real r as foreign "JAVA:MyForeignFunctions/sqroot";
```

The function is now defined for the system. The query optimizer assumes a very low cost for foreign functions, In case the foreign function returns bag of values (Sec. 3.2.2) or is expensive to execute you can define your own cost model by using *cost hints* (Sec. 4.).

3.2.2   Bag-valued foreign functions

An Amos II function can return a bag (stream) of values. For example the sqroot function could return both the positive and the negative roots. This is achieved by calling emit several time, e.g.:

```
import callin.*;
import callout.*;
public class MyForeignFunctions
{
        public void sqroot(CallContext cxt, Tuple tpl) throws AmosException
        {
                double x;

                x = tpl.getDoubleElem(0);// Pick up the argument
                if (x == 0.0) // One root
                {
                        tpl.setElem(1,0.0);
                        cxt.emit(tpl);
                }
                else if (x > 0.0) // two roots
                {
                        double r = Math.sqrt(x);

                        tpl.setElem(1, r);
                        cxt.emit(tpl);
                        tpl.setElem(1, -r);
                        cxt.emit(tpl);
                }
                // negative numbers have no roots
        }
}
```

3.2.3 Multidirectional Foreign Functions

Amos II functions can be *multi-directional*, i.e. they can be executed also when the result of a function is given and some corresponding argument values are sought. For example, if we have a function

parents(person)-> bag of person

we can ask these AMOSQL queries:

```
parents(:p);  /* Result is the bag of parents of :p */
select c from person c where parents(c) = :p;
        /* Result is bag of children of :p */
```

It is often desirable to make Foreign Amos II functions invertible as well. For example, we may wish to ask these queries using the square root function sqroot:

```
sqrt(4.0);   /* Result is -2.0 and 2.0 */
select x from number x where sqrt(x)=4.0;
        /* result is 16.0 */
```

With simple foreign Amos II functions only the first function call is possible. *Multi-directional foreign functions* permit also the second kind of queries.

Multi-directional foreign functions are functions that can be executed when some results are known rather than just the arguments. The benefit of multi-directional foreign functions is that a larger class of queries calling the function is executable (safe), and that the system can make better query optimization. A multi-directional foreign function can have several implementations depending on the *binding pattern* of its arguments and results. The binding pattern is a string of b:s and f:s, indicating which arguments or results in a given implementation are known or unknown, respectively.

A simple foreign AMOS function is a special case where all arguments are known and all results are unknown. For example the binding pattern of sqroot above is the list "bf".

The square root actually has the following possible binding patterns:

(1) If we know X but not R and X>=0 then R = sqrt(X) and R = -sqrt(X)
(2) If we know R but not X and R then X = R**2
(3) If we know both R and X then check that sqrt(X) = R

Case (1) is implemented by the simple foreign function sqroot above.

Case (3) need not be implemented as it is inferred by the system by first executing sqroot(X) and then checking that the result is equal to R (see [2]).

However, case (2) cannot be handled by the simple foreign function sqroot as it requires the computation of the inverse function (square) rather than the square root. Case (2) can be implemented as a multi-directional foreign function and we will now specify a generalized sqroot using that mechanism.

To implement a multi-directional foreign function you first need to think of for which binding patterns implementations are needed. In the sqroot case one implementation handles the square root and the other one handles the square. The binding patterns will be "bf" for the square root and "fb" for the square. The Java method sqroot above implements only the "bf" case. The "fb" case is implemented by the following method, square:

```
public void square(CallContext cxt, Tuple tpl) throws AmosException
{
        double x;

        x = tpl.getDoubleElem(1);// Pick up the result
        tpl.setElem(0,x*x);  // compute the inverse of the square root
        cxt.emit(tpl);
}
```

A multi-directional foreign function is created by executing the AMOSQL statement:

```
create function <fn>(<argument declaration>) -> <result declaration>
        as multidirectional ('<bpat> foreign 'JAVA:<class file>/<method>')...;
```

For example:

```
create function sqroot(real x) -> real
        as multidirectional
            ('bf' foreign 'sqroot')('fb' foreign 'square');
```

As for simple foreign function definitions, <fn> is the name of the foreign function, <argument declaration> is the signature of its argument(s), and <result declaration> is the signature of its result(s). For each binding pattern <bpat> a corresponding foreign function definition in Java, 'JAVA:<class file>/<method>', is specified.

Separate cost hints can be assigned to each binding pattern (Sec. 4.).

3.2.4   Exception Handling in Foreign Functions

Amos II can raise two kinds of Java Acxeptions: NoMoreData and AmosException.

NoMoreData is raised by the a_emit when the consumer of the result from a foreign function iterating over a scan does not need any more data, i.e. the scan is *terminated*. When a scan is terminated the system throws the exception NoMore-Data from a_emit.The user need not declare or catch NoMoreData in foreign functions. However, often a try ... finally ... construct is needed in order to guarantee that resourses allocated by the foreign function are always freed.

AmosException is raised by the sytem when some error is detected. The standard error message for an Amos II exception can be obtained by calling the Java exception method getMessage.

The user can also raise AmosException to indicate new errors. To raise a new Amos II exception with a new error message, AmosException has a constuctor that takes an error string as parameter.

The cause of an AmosException can be investigated through the following AmosException attributes:

errno: Attribute holding the AMOS II error number. Only some of the Amos II system messages have error number; if errno==-1 it indicates that the exception did not have a specific error number.

errstr: Attribute holding the AMOS II error message string.

errform: Attribute holding the AMOS II Oid causing the error.

4.   Cost Hints

Different implementations of multi-directional foreign functions often have different execution costs. In the sqroot example the cost of computing the square root is higher than the cost of computing the square. Therefore the query optimizer needs *cost hints* that help it choose the most efficient implementation. In the example we might want to indicate that the cost of executing a square root is double as large as the cost of executing a square.

Furthermore, the query optimization is also very much dependent on the expected size of the result from a function call. This is called the *fanout* (or selectivity) of the call for a given binding pattern. In the sqroot example the implementation usually has a fanout of 2, while square has a fanout of 1.

Multi-directional function definitions may include cost hints, for example:

```
create function sqroot(real x)->real y as
        multidirectional
                ("bf" foreign "JAVA:MyForeignFunctions/sqroot" cost {4,2})
                ("fb" foreign "JAVA:MyForeignFunctions/square" cost {2,1});
```

In this case the vector {4,2} indicates that the cost of executing the method sqroot is 4 and it returns a stream (bag) of two values. By contrast the cost of executing square is 2 and it returns a single value.

Notice here that the cost and fanouts are used by the query optimizer for comparing costs when it needs to choose between two possible equivalent execution plans for a query. It is therefore not necessary to have a very careful cost model; the most important thing is that it can discriminate between alternative implementations reasonably. As a rough reference point for the cost model, the system assumes the cost 1 of accessing a hash table in main memory.

For example, if the user specifies the query

        select true where sqroot(4.0)=2.0;

the system will call the method square, not sqroot, since it is cheaper.

In the example above the costs and the fanout were constants. However, it is often not possible to specify these numbers as constants. Therefore the cost hints can be computed dynamically as a function, rather than as a constant vector of numbers. Notice that the cost hint function is normally not called at run time when the query execution plan is interpreted, but at compile time, when the execution plan is generated. A dynamic cost hint function has the signature:

create function <hintfn>(function f, vector bpat, vector argl)
      -> <integer cost, integer fanout> ...

The arguments are:

f   the called AMOS function.

bpat   is the binding pattern of the call as a vector of strings 'b' and 'f', e.g. {"f","b"}, indicating which arguments in the call are bound or free, respectively.

argl   Not accessible from Java.

The cost hint function returns two results:

cost   is the computed estimated cost to execute a call to f with the given binding pattern and argument list. The cost to access a tuple of a stored function (by hashing) is 2; other costs are calibrated accordingly.

fanout   is the estimated fanout of the execution, i.e. how many results are returned from the execution.

If the cost hint function does not return anything it indicates that the function is not executable for the given binding pattern.

References

1   S.Flodin, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, and M.Werner: *Amos II User's Manual,* UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden
    http://www.dis.uu.se/~udbl/amos/doc/amos_users_guide.html

2   W.Litwin, T.Risch: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, in *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

3   T.Risch: *Amos II External Interfaces*, UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden
    http://www.dis.uu.se/~udbl/amos/doc/external.pdf

4   T.Risch: *ALisp User's Guide*, UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden
    http://www.dis.uu.se/~udbl/amos/doc/alisp.pdf

5   T.Risch, V.Josifovski, T.Katchaounov: *AMOS II Concepts*, UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden
    http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html

6   Guy L.Steele Jr.: *Common LISP, the language*, Digital Press,
    http://www.ida.liu.se/imported/cltl/cltl2.html

7   M.Stonebraker: *Object-Relational Databases*, Morgan Kaufmann, 1996.