Amos II External Interfaces

Tore Risch
Uppsala Database Laboratory
Department of Information Technology
Uppsala University
Sweden
Tore.Risch@it.uu.se

2001-01-24
(Latest revision 2012-12-19)

This report describes the external interfaces between Amos II and the programming languages C/C++ and Lisp. There is also an interface between Amos II and Java documented separately. There are two ways to interface Amos II with other programs: Either an external program calls Amos II through the *callin* interface, or foreign AmosQL functions are defined by external subroutines through the *callout* interface. The combination is also possible where external subroutines call Amos II back through the callin interface.

1. Introduction

There are external interfaces between Amos II and the programming languages ANSII C, Lisp, and Java. The low level external C interfaces are intended to be used by systems developers who need to access and extend the kernel of the system. This document describes the C and Lisp interfaces, while the Java interfaces are documented separately. The Java interface is the most convenient way to write Amos II applications and to do simple extensions. The Lisp and C interfaces must be used for more advanced system extensions or time critical applications.

Amos II contains an interpreter for ALisp [3] which a subset of CommmonLisp [4]. Much internal functionality in Amos II is written in ALisp. However, you do not need to know ALisp in order to use the external C interfaces or Java interfaces of Amos II.

This report first describes the external interfaces with ANSII C, which are the most basic external interfaces. Then the ALisp interfaces are described.

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

- With the *callin* interface a program in C calls Amos II. The callin interface is similar to the call level interfaces for relational databases, such as ODBC, JDBC, SYBASE, ORACLE call-level interface, etc.
- With the *callout* interface AmosQL functions call external subroutines written in C (Lisp, Java). These *foreign AmosQL functions* are implemented by a number of C functions. The callout interface has similarities with 'data blades' in Object-Relational databases [5]. The foreign functions in Amos II are *multi-directional* which allow them to have separately defined capabilities and to be indexed [1][2]. The system furthermore allows the callin interface to be used in the foreign functions, which gives great flexibility and allow them to be used as a form of stored procedures.

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

- In the *embedded query* interface strings containing AmosQL statements are passed to Amos II for evaluation. Primitives are provided to access the results of the AmosQL statements from C. The embedded query interface is relatively slow since the AmosQL statements have to be parsed and compiled.
- In the *fast-path* interface predefined Amos II functions are called from C, without the overhead of 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.

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

- Amos II can be linked directly with an application program in C. 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 errors in the application program may cause Amos II to crash. Another disadvantage is that only a single application (but many threads) can be linked to Amos II. This normally means that Amos II becomes a single-application system.
- The 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 applications and the Amos II server run as different programs. Application crashes will not bring down the Amos II server. However, the overhead of calling Amos II from another program can be several hundred times slower than the tight connection.

The query language AmosQL and the language independent primitives for defining multidirectional foreign function is documented in [1]. The callout interface can be used, e.g., for implementing customized data structures in C that are made fully available in Amos II. The callout interface always runs in the same address space. To fully utilize the power of the callout interface requires good knowledge of the internals of the Amos II storage management. However, it is the intention of the C callout interface that it should be fairly easy to extend Amos II with simple external AmosQL functions.

It is often simpler to use the callout interface to ALisp or Java than to C. Those interfaces protect the user from doing illegal memory operations and the languages have built-in garbage collectors. However, callout to C is required when the performance of ALisp or Java is insufficient or when external C libraries need to be called from Amos II.

This document is organized as follows:

- Sec. 2. documents the *callin* interface to Amos II from C (Sec. 2.1) or ALisp (Sec. 2.2) using either the embedded query or the fast-path interface.
- Sec. 3.describes the *callout* interface where Amos II functions can be defined in C (Sec. 3.1) or ALisp (Sec. 3.2).

2. The Callin Interface

This section describes how to call AmosQL from C and ALisp using the embedded query or the fast-path interface.

2.1 Calling Amos II from C

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

- There exists both an *embedded query* interface (Sec. 2.1.4) and a *fast-path* interface (Sec. 2.1.8) from C to Amos II. The fast-path interface is up two orders of magnitude faster.
- There is a *tight connection* between an application and Amos II where Amos II is linked to the application program. A C library, amos2.lib (Borland C++) or amoslib.lib (MicroSoft C++), is provided for this. There is also a DLL, amos2.dll which is currently used only by Borland C++. Static linking is used for MicroSoft C++ and Unix.
- In order to call Amos II from C a *driver program* (Sec. 2.1.1) is needed. It is a C main program that sets up the Amos II environment and then calls the Amos II system. Fatal errors raised in Amos II should be trapped by the driver program.
- Data is passed between the application and Amos II using some special C data structures called *handles*.

The system is delivered with a sample program which demonstrates how to call Amos II from C, `amosdemo.c`.

2.1.1 The driver program

This is an example of a driver program with some declarations of Amos II *handles*:

```
#include "callin.h" /* Include file when calling Amos II from C */
main(int argc, char **argv)
{
       dcl_connection(c); /* c is a connection handle */
       dcl_scan(s);  /* s is a scan (stream) handle from results of Amos II queries and
                     function calls */
       dcl_tuple(row);  /* row is a tuple handle */
       dcl_oid(o); /* o is a handle to an Amos II object */

       /*** Put your other declarations here ***/

       init_amos(argc,argv); /* Initialize embedded Amos II
                                with command line parameters */
       /*** Put your bindings of foreign functions to C-code in embedded Amos II here
(Sec. 3.1.3) ***/
```

```
        /* Connect to Amos II peer or to embedded Amos II database (name ""): */
        a_connect(c,"",FALSE);

        /*** Put your application code here ***/

        a_disconnect(c,FALSE); /* Close the connection */
        free_tuple(row); /* Free row handle */
        free_oid(o); /* Free object handle */
        free_scan(s); /* Free scan handle */
        free_connection(c); /* Free connection handle */
};
```

The system is initialized based oncommand line parameters with the C function:

```
        init_amos(int argc, char **argv)
```

`init_amos` knows how to handle command line parameters. For example, a database image or an AmosQL initialization script may be specified. The execution of initialization scripts is delayed until the first call to `a_connect` (Sec. 2.1.3) to allow for foreign function initializations (Sec. 3.1.8.3) to be made before the initialization script is loaded.

If Amos II is embedded in other systems it is often better to initialize Amos II without access to the command line parameters, without having a driver main program, and with the possibility to trap initialization errors. You can then initialize the system with:

```
        int a_initialize(char *image, int catcherror)
```

`a_initialize` initializes the system with the specified image. `a_initialize` returns 0 if the initialization was successful. If the initialization failed and `catcherror` is FALSE the system will exit. If `catcherror` is TRUE the error is trapped, an error number returned, and the error can be handled according the error description mechanisms in Sec. 2.1.2.

To enter the interactive AmosQL top loop from C, call

```
        void amos_toploop(char *prompter)
```

`amos_toploop` returns if the AmosQL command '`exit;`' is executed. By contrast, the AmosQL command '`quit;`' terminates the program.

2.1.2 Catching errors

Most Amos II C interface functions have the parameter `catcherror` that normally should be FALSE, indicating that if the call fails it should cause a fatal error which is trapped by the system error trap. For example:

```
        int a_initialize(char *image, int catcherror);
```

The interface functions normally return an *error indication* which is FALSE (=0) if the call was OK and an *error number* if an error occurred and `catcherror` is TRUE. The error can then be investigated by looking at these global C variables:

```
        int a_errno; /* The Amos II error number, same as the error indication */
        char *a_errstr; /* A string explaining the error */
        oidtype a_errform; /* A reference to an Amos II object */
```

Some interface functions below do not return the error indication. The occurrence of an error must then be investigated by accessing `a_errno`.

2.1.3 Connections

The C macro

```
dcl_connection(c);
```

allocates a *connection handle* to an Amos II database and binds it to the specified C variable of type `a_connection`. To actually connect to an Amos II server the connection must be established with the call:

```
int a_connect(a_connection c, char *peer, int catcherror);
```

`peer` is the name of the Amos II database to connect to. If `peer` is the empty string it represents a connection to the embedded database; otherwise `peer` must be the name of an Amos II peer known to the nameserver running on the *same* host as the application.

If the nameserver resides on host `hostid` the connection can be made with the call:

```
int a_connectto(a_connection c, char *peer, char *hostid, int catcherror);
```

NOTICE that if `a_connectto` is called several times the `hostid` must be the same!

The loading of initialization scripts specified on the command line when the system is initialized with `init_amos` is delayed until the first call to `a_connect` or `a_connectto`.

Every connection must always be terminated when no longer used with the call:

```
int a_disconnect(a_connection c, int catcherror);
```

Finally the connection object must always be deallocated when no longer used by calling:

```
void free_connection(a_connection c);
```

`free_connection` will actually do an `a_disconnect` too, but without error trapping.

2.1.4 The Embedded Query Interface

In the *embedded query* interface, strings with AmosQL statements can be executed from C by calling

```
int a_execute(a_connection c, a_scan s, char *string, int catcherror);
```

For example

```
a_execute(c,s,"select name(t) from type t;",FALSE);
```

The string is a C string with an AmosQL statement ended with';'. The result of the query is stored in a *scan* associated with the connection `c`. It can be accessed as explained below.

NOTICE that the cost of optimizing query strings can be very high and it is therefore recommended to use the fast path interface (Sec. 2.1.8) when possible.

2.1.4.1 Scans

In the embedded query interface, or when Amos II functions are called, the results are always returned as *scans*. A scan is a stream of *tuples* which can be iterated through with special interface functions (Sec. 2.1.6). A *scan handle* `s` (C type `a_scan`) is allocated with:

```
dcl_scan(s);
```

Every scan handle `s` must be freed when no longer used with:

```
free_scan(s);
```

The returned scans contain tuples. A *tuple handle* tpl (C type `a_tuple`) is allocated with:

```
dcl_tuple(tpl);
```

Every no longer used tuple handle tpl must be released with:

```
free_tuple(tpl);
```

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

```
dcl_tuple(tpl); /* Allocate tuple handle */
...
while(!a_eos(s)) /* While there are more rows in scan */
{
       a_getrow(s,tpl,FALSE); /* Get current tuple in scan */
       ... code to retrieve elements of tuple tpl ...
       a_nextrow(s,FALSE); /* Advance scan forward */
}
```

The following C functions are used to iterate through scans:

```
a_eos(a_scan s)
```

returns TRUE if there are no more tuples in the scan s.

```
int a_getrow(a_scan s, a_tuple tpl, int catcherror)
```

copies a reference to the current tuple in the scan s into the tuple handle tpl.

```
int a_nextrow(a_scan s, int catcherror)
```

advances the scan s forward, i.e. sets its current tuple to the next tuple in the scan.

The scan is automatically *closed* by the system when all tuples have been read or if it is assigned to another query result. In case you for some reason need to close the scan prematurely you can close it explicitly by calling:

```
int a_closescan(a_scan s, int catcherror)
```

2.1.4.2 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:

```
a_execute(c,s,"select t from type t where name(t)='person';",FALSE);
a_getrow(s,tpl,FALSE);
/* The tuple is now available in tpl without any need to open s scan */
```

2.1.5 Object Handles

Object handles are references to Amos II database objects from C. The object handles can reference any kind of data stored in Amos II databases, including number, strings, Amos II OIDs, arrays, and other internal Amos II data structures.

Object handles must be allocated with:

```
dcl_oid(o);
```

Every object handle o must be deallocated when no longer used:

```
free_oid(o);
```

An object handle is basically a logical pointer to an Amos II data structure. Object handles have the C datatype `oidtype`. They are currently represented as unsigned integers. Amos II data are always accessed through these handles. The object handles are initialized by `dcl_oid` to a system handle, `nil`. Some conventions must be followed for a well behaved application program.

NOTICE: Handles should always be initialized using `dcl_oid`. If the handles are uninitialized the system might crash.

A handle referenced from a location can be assigned a new value by the `a_assign` function:

```
a_assign(<location>,<new value>)
```

For example:

```
a_assign(fn,a_getfunction(c,"charstring.typenamed->type",FALSE));
```

An incremental garbage collector based on reference counting is invoked by `a_assign` so that the old value stored in <location> is deallocated if *no other* location references the old value.

WARNING: Do not use C's assignment operator for object handles (unless you really know what you are doing). Always use `dcl_oid` and `a_assign` instead!

You can print an object handle with

```
oidtype a_print(oidtype x);
```

The result is x itself. The printing will indicate what kind of Amos II object is referenced from the handle x.

Object must be deallocated by

```
free_oid(<location>)
```

The system will then deallocate the object referenced by the handle only if no other location still references it. There will be memory leaks if you don't call `free_oid` when you no longer need a location (e.g. when you exit the C block where the location is declared).

2.1.6 Tuples

Tuples are used when passing data from C to Amos II and vice versa. There are several system functions for manipulating tuples. Tuple handles are allocated with

```
dcl_tuple(tpl);
```

and must be deallocated with

```
free_tuple(tpl);
```

The following code, printing the names of all types in an Amos II database, illustrates how tuples are used when re-

trieving data from Amos II:

```
a_execute(c,s,"select name(t) from type t;",FALSE); /* FALSE => no error trapping */
  /*          s is a 'scan' holding the result of the query */
  while(!a_eos(s)) /* While there are more rows in scan */
  {
     char str[50];
     a_getrow(s,tpl,FALSE); /* Get current tuple in scan */
    a_getstringelem(tpl,0,str,sizeof(str),FALSE); /* Get 1st elem in tuple as string */
     printf("Type %s\n",str);
     a_nextrow(s,FALSE); /* Advance scan forward */
  }
```

As shown in the example, tuples are used to retrieve data from results of Amos II commands and function calls. Tuples are furthermore used for representing argument lists in Amos II function calls (Sec. 2.1.8), for representing sequences (Sec. 2.1.6.5), and for interfacing foreign Amos II functions defined in C (Sec. 3.1). There are a number of system functions for moving data from the tuples to C application data areas. It is also possible to copy data from C application data areas into tuples.

The function

```
int a_getarity(a_tuple t, int catcherror);
```

returns the number of elements in (width of) the tuple t. The elements of a tuple are enumerated starting at 0 and can be accessed through a number of tuple access functions to be described next.

To allocate a new tuple with the a given arity and store it in a tuple handle tpl, use:

```
a_newtuple(a_tuple tpl, int arity, int catcherror);
```

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

For debugging purposes a tuple `tpl` can be printed on standard output with

```
a_print(tpl->tpl);
```

2.1.6.1 String elements

To copy data from an element in a tuple into a C string use the function:

```
int a_getstringelem(a_tuple t, int pos, char *str, int maxlen, int catcherror);
```

It copies element number `pos` of the tuple `t`, that must be a string, into a C character string `str` whose maximum length is `maxlen`. An error occurs if the value in element number `pos` is not a string.

To copy data from a C string str into element `pos` of a tuple `t` use the function:

```
int a_setstringelem(a_tuple t, int pos, char *str, int catcherror);
```

2.1.6.2 Integer elements

To get an integer stored in position `pos` of the tuple `t` use the function:

```
int a_getintelem(a_tuple t,int pos, int catcherror);
```

It returns the integer (not the error indication). An error is generated if there is no integer in the specified position of the tuple.

To store an integer `v` in element `pos` of a tuple `t` use the function:

```
int a_setintelem(a_tuple t,int pos, inv v, int catcherror);
```

2.1.6.3 Floating point elements

To get a double precision floating point number stored in position `pos` of the tuple `t` use the function:

```
double a_getdoubleelem(a_tuple t, int pos, int catcherror);
```

It returns the number (not the error indication). An error is generated if there is no a real number in the specified position of the tuple.

To store a floating point number `v` in element `pos` of a tuple `t` use the function:

```
int a_setdoubleelem(a_tuple t, int pos, double v, int catcherror);
```

It returns the error indication.

2.1.6.4 Object elements

To get an object handle stored in position `pos` of the tuple `t` use the function:

```
oidtype a_getobjectelem(a_tuple t, int pos, int catcherror);
```

It returns an Amos II *object handle* (not the error indication). `nil` is returned if an error occurred. An Amos II object handle identifies any kind of data structure supported by the Amos II storage manager, as explained in Sec. 2.1.5.

To store an object handle in element `pos` of a tuple `t` use the macro:

```
int a_setobjectelem(a_tuple t, int pos, oidtype o, int catcherror);
```

2.1.6.5 Sequences

Sequences are special Amos II objects which hold references to several other objects. Data values of type *Vector* in AmosQL represent sequences in Amos II databases. Sequences are represented as *tuples* in the external C interface.

The following function moves into the tuple handle `seq` the sequence stored in position `pos` of the tuple `t`:

```
int a_getseqelem(a_tuple t, int pos, a_tuple seq, int catcherror);
```

An error is generated if there is no sequence (tuple) in the specified position of the tuple.

The elements of the sequence can be investigated using the regular tuple accessing primitives.

To store a copy of the tuple `seq` as a sequence in element `pos` of the tuple `t` use the function:

```
int a_setseqelem(a_tuple t, int pos, a_tuple seq, int catcherror);
```

2.1.7 Data type tests

Each object handle has an associated *data type* which specifies what kind of object the handle references. Some of the data types also require an associated *size* for each object handle.

The data type of an object handle o can be inspected with the macro:

```
int a_datatype(oidtype o)
```

`a_datatype` returns a *type tag* which is an integer indicating the data type of the object referenced from o. The following built-in data types, defined as C macros, are of interest:

INTEGERTYPE denotes integers.

REALTYPE denotes double precision floating point numbers.

STRINGTYPE denotes strings.

ARRAYTYPE denotes sequences.

SURROGATETYPEdenotes references to object handles representing Amos II objects.

To get the data type of an element in a tuple use the macro:

```
int a_getelemtype(a_tuple tpl, int pos, int catcherror)
```

It returns the type tag of the element at position `pos` of the tuple `tpl`.

Every object handle also has an associated *size*. For strings and sequences the size can vary. To get the size of an element in a tuple use the function:

```
int a_getelemsize(a_tuple tpl, int pos, int catcherror)
```

2.1.8 .The Fast-Path Interface

The *fast-path* interface permits Amos II functions to be called from C. Primitives are provided for binding argument lists to C data. The result of a fast-path function call is always as scan that is treated in the same way as the result of an embedded query call (Sec. 2.1.4).

The following is an example shows how to call the Amos II function `charstring.typenamed->type` from C:

```
dcl_tuple(argl); /* Allocate tuple handle to hold argument lists */
.....
/* Get a handle to the function to call */
a_assign(f1,a_getfunction(c,"charstring.typenamed->type",FALSE));
a_newtuple(argl,1,FALSE); /* There is one argument */
a_setstringelem(argl,0,"FUNCTION",FALSE); /* Bind string to first argument */
a_callfunction(c,s,f1,argl,FALSE); /* Call the function */
a_getrow(s,row,FALSE); /* Move single scan element to row */
a_print(a_getobjectelem(row,0,FALSE));
                                  /* Print the single result Amos II object */
.....
free_tuple(argl);
```

The following functions and macros are used for setting up fast-path function calls:

```
a_getfunction(a_connection c, char *fnname, int catcherror);
```

`a_getfunction` returns the function object named `fnname` (not the error indication) or `nil` if no such functions exists.

NOTICE that you have to use `a_assign` to bind the returned Amos II object handle to a C location.

2.1.8.1 Argument lists

Argument lists represent arguments to be passed to fast-path Amos II function calls. They are represented as *tuples*, and thus declared as

```
dcl_tuple(argl);
```

Before the Amos II function is called the argument list tuple must be allocated with the correct arity of the function to call with

```
a_newtuple(a_tuple argl, int arity, int catcherror);
```

The elements of the argument list should then be assigned with some tuple update function (Sec. 2.1.6). The first element in the tuple (position 0) is the first argument, etc.

When all arguments in the argument list are assigned the Amos II function can be called with

```
int a_callfunction(a_connection c, a_scan s, oidtype fn, a_tuple args,
                   int catcherror);
```

where c, s, and catcherror have the same meaning as for `a_execute`. fn is the handle of the function to call and args is the argument list tuple.

2.1.9 Creating and Deleting Objects

New Amos II objects are created with

```
oidtype a_createobject(a_connection c, oidtype type, int catcherror);
```

where c is a connection, type is the type of the new object, and catcherror is the error trapping flag. a_createobject returns the newly created object (not the error indication) or nil if there was an error.

To get an object handle to the object representing a type named name use

```
oidtype a_gettype(a_connection c, char *name, int catcherror);
```

where c is a connection and catcherror is the error trap flag. a_gettype returns the type named name (not the error indication). If catcherror is set to TRUE and no type was found, nil is returned and a_errorflag is set to TRUE.

To delete an object o use

```
int a_deleteobject(a_connection c,oidtype o,int catcherror);
```

2.1.10 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 update functions from C:

```
dcl_tuple(argl1);
dcl_tuple(resl2);
dcl_oid(fn);
a_newtuple(argl1,1,FALSE); /* Argument tuple holding 1 argument */
a_newtuple(resl2,2,FALSE); /* Result tuple holding 2 result values */
.....
/*** Create an populate new stored function salary ***/
a_execute(c,s,
       "create function salary (charstring name)-> <integer s, real sc>;",FALSE);
a_assign(fn,a_getfunction(c,"charstring.salary->integer.real",FALSE));
/* It would have worked with this 'generic' getfunction too:
       a_assign(f2,a_getfunction(c,"salary",FALSE));
       but it would have been radically slower! */
a_setstringelem(argl1,0,"Tore",FALSE);
a_setintelem(resl2,0,1000,FALSE);
a_setdoubleelem(resl2,1,3.4,FALSE);
a_setfunction(c,f2,argl1,resl2,FALSE);
/* 2nd new value */
a_setstringelem(argl1,0,"Kalle",FALSE);
a_setintelem(resl2,0,2000,FALSE);
a_setdoubleelem(resl2,1,2.3,FALSE);
a_addfunction(c,f2,argl1,resl2,FALSE);
.....
free_tuple(argl1);
free_tuple(resl2);
free_oid(fn);
```

To assign a new value to an Amos II function use:

```
int a_setfunction(a_connection c, oidtype fn, a_tuple argl, a_tuple resl,
                            int catcherror);
```

where c is a connection, fn is a function handle, argl is the argument tuple, resl is the result tuple, and catcherror is the error trapping flag. a_setfunction assigns resl as the result of applying fn on the arguments argl.

To add a new tuple to the values of a bag valued function use:

```
int a_addfunction(a_connection c, oidtype fn, a_tuple argl, a_tuple resl,
                          int catcherror);
```

The arguments and result of `a_addfunction` are the same as for `a_setfunction`. For bag valued functions `a_addfunction` adds a result tuple while `a_setfunction` deletes the old result tuples and replace them with a single new tuple `resl`.

To remove a result tuple from a bag of result tuples for a given argument tuple, use:

```
int a_remfunction(a_connection c,oidtype fn, a_tuple argl, a_tuple resl,
                          int catcherror);
```

2.1.11 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.

To commit a transaction use:

```
int a_commit(a_connection c,int catcherror);
```

To rollback (abort) a transaction use:

```
int a_rollback(a_connection c,int catcherror);
```

2.2 Calling Amos II from ALisp

To call AmosQL from ALisp use the Lisp macro

```
(OSQL string)
```

where string is an arbitrary AmosQL statement ended with a ';'. Environment variables in AmosQL statements are translated into Lisp variables prefixed with AMOS_.

For example:

```
(defun printname (AMOS_o)
       (let (AMOS_nm)
              (osql "select name(:o) into :nm;")
              (print AMOS_nm)))
```

This *embedded query* interface from ALisp can be very efficient since it is based on Lisp macros that translate the AmosQL statements to fast Lisp code. However, avoiding inefficient late binding in calls to AmosQL from ALisp requires *type declared* interface variables. The following macro globally declares an ALisp variable to be of a specific Amos II type:

```
(osql-declare <type> <variable>)
e.g. (osql-declare person :o)
```

It is the responsibility of the programmer to make sure that a declared variable actually gets bound to the declared type.

The global declaration of a Lisp interface variable can be removed with:

```
(osql-undeclare <variable>)
e.g. (osql-undeclare :o)
```

To locally bind and declare ALisp variables use:

```
(osql-let ((<type> <variable>) ...)
  ...)
e.g. (defun printname (p)
       (osql-let ((person :o) (charstring :nm))
              (setq AMOS_o p) ; notice how to declare function parameters!
              (osql "select name(:o) into :nm;")
              (print AMOS_nm)))
```

NOTICE: Do not call `osql-declare` inside `osql-let` since the declaration then will be local and thus removed when the `osql-let` is exited. `osql-declare` is intended to be used outside `osql-let` to declare global interface variables.

The ALisp-AmosQL interface is optimized for flat Amos II function calls. Nested queries and ad hoc select statements will cause dynamic optimization every time the OSQL form is executed which is *very* slow. For example, the following code is very slow:

```
(defun printname (p)
       (osql-let ((person :o) (charstring :nm))
              (setq AMOS_o p)
              (osql "select nm into :nm for each charstring nm where
                           nm = name(:o);")
              (print AMOS_nm)))
```

3. The Callout Interface

This section explains how to implement Amos II functions in C and ALisp through the callout interface. AmosQL functions can be defined in C or ALisp through a special mechanism called the *multi-directional foreign function* interface [1][2]. Foreign functions used in AmosQL queries should be side-effect free since the query optimization may rearrange their calling sequence.

We first describe the simplest case, where an AmosQL function implemented in C 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]. Finally we show how to define new aggregation operators, and logical operators, such as negation of sub-queries.

The demonstration program `calloutdemo.c` illustrates how to define AmosQL functions in C.

3.1 Implementing Amos II functions in C

A simple foreign Amos II function computes its result given a set of actual arguments represented as a *tuple*, similar to a subroutine in a regular programming language. The driver program must contain the following code to define foreign functions:

1. C code to *implement* the function.
2. A *binding* of the C implementation to a *symbol* in the Amos II database.
3. A *definition* of the foreign AmosQL function.
4. An optional *cost hint* to estimate the cost of executing the function.

3.1.1 Function Implementation

A foreign function implementation fn in C has the signature:

```
void fn(a_callcontext cxt, a_tuple tpl);
```

where `cxt` is an internal Amos II data structure for managing the call and `tpl` is a tuple representing actual arguments and results. The argument values are first, followed by the unassigned result values of the function. For example, the following C function implements the square root:

```
#include "callout.h" /* always include callout.h when defining foreign functions */
void sqrtbf0(a_callcontext cxt, a_tuple tpl)
{
	double x;

	x = a_getdoubleelem(tpl,0,FALSE); /* Pick up the argument */
	if(x>=0.0) /* Do not return any value if sqrt undefined */
	{
		a_setdoubleelem(tpl,1,sqrt(x),FALSE); /* Set value to sqrt(x) */
		a_emit(cxt,tpl,FALSE);
	}
	return;
}
```

Notice that `callout.h` must be included.

The size of the tuple `tpl` is the sum of the number of arguments of the Amos II function (its arity) and the width of its result tuple.

The C function emits result tuples of the same size as tpl with the result variable position(s) filled in. Therefore in the example the position 1 of tuple tpl is set to the resulted square root.

The result tuple tpl is returned to Amos II through the function

```
int a_emit(a_callcontext cxt, a_tuple tpl, int catcherror);
```

where cxt is the cxt from the actual arguments of the function definition. `a_emit` returns the error indication.

Notice that a result tuple of `a_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 `a_emit` is not called the result of the foreign function will be empty (nil). In the example above no result is returned when the argument is a negative number.

NOTICE that, for efficiency, the parameter tuple tpl is temporarily allocated on the C stack and can therefore NOT be saved somewhere.

3.1.2 Bag-valued foreign functions

An AmosQL function can return a bag of values (actually a stream of values). For example the square root function could return both the positive and the negative square root. This is achieved by calling `a_emit` several time, e.g.:

```
void sqrtbf(a_callcontext cxt, a_tuple tpl)
{
	double x;

	x = a_getdoubleelem(tpl,0,FALSE); /* Pick up the argument */
	if(x<0.0) /* No root */
		return;
	if(x==0.0) /* One root 0.0 */
```

```
	{
		a_setdoubleelem(tpl,1,0.0,FALSE);
		a_emit(cxt,tpl,FALSE);
	}
	else if(x>=0.0) /* Two roots */
	{
		a_setdoubleelem(tpl,1,sqrt(x),FALSE);
		a_emit(cxt,tpl,FALSE);
		a_setdoubleelem(tpl,1,-sqrt(x),FALSE);
		a_emit(cxt,tpl,FALSE);
	}
	return;
}
```

3.1.3 Giving names foreign function implementations

Before Amos II can use a foreign function implementation, the implementation must be associated with a symbolic name inside Amos II. The driver program must therefore bind the C function to a string. This is done with:

```
	a_extfunction(char *name,external_predicate fn);
```

where name is an identifier for the foreign function and fn is a pointer to the C function.

For example:

```
	a_extfunction("sqrtbf",sqrtbf);
```

The naming is normally made before the first call to `a_connect`.

3.1.4 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 declarations>) -> <result declaration>\
	as foreign '<name>';
```

where <fn> is the above symbolic name of the foreign function, <argument declarations> is the signature of its arguments, <result signature> is the signature of its results, and <name> is the name of the C implementation as specified in the binding. The definition can be done through `a_execute` in C or by AmosQL commands. The definition needs only be done once and then saved with the image.

For example:

```
create function sqrt(real x) -> real as foreign 'sqrtbf';
```

The function is now defined for the system. The query optimizer assumes a very low cost for foreign functions and on the average slightly more than one tuple returned. In case the foreign function is expensive to execute or returns large bags of values, you can define your own cost model by using multidirectional foreign functions having cost hints [1].

3.1.5 The driver program

One thing to notice for the driver program is that symbolic bindings must be executed *after* the initialization of Amos II but *before* the first `a_connect`. The complete driver program, called myApp, looks like this:

```
#include "callout.h"
main(int argc,char **argv)
{
```

```
        dcl_connection(c); /* To hold connection to Amos */

        init_amos(argc,argv); /* Initialize embedded Amos */
        /* Bind C function 'sqrtbf' to Amos symbol 'sqrtbf': */
        a_extfunction("sqrtbf",sqrtbf);
        a_connect(c,"",FALSE); /* Connect to embedded Amos */
        /* Define Amos II function 'sqrt(real x) -> real' as sqrtbf
               (can also be done separately in a script as explained below): */
        a_execute(c,s,"create function sqrt(real x) -> real as foreign 'sqrtbf';",
               FALSE);
        amos_toploop("Amos"); /* Enter interactive Amos II top loop */
        free_connection(c);
        exit(0);
}
```

The program can be started with the command

```
myApp amos2.dmp
```

where ’amos2.dmp’ is the initial system image.

The resolvent definition is saved in the database image if a ’save’ AmosQL command is issued. If our program is started from such a saved database image the resolvent will be (re)defined every time the program is run, even though the definition was already in the image. To avoid this one can place all foreign function resolvent definitions in a special AmosQL script instead of defining them by using `a_execute` as in the C-code as above.

For example, we may remove the `a_execute` call and put the following AmosQL code in a file ’`init.amosql`’, which is used only once when the initial database image is created from the system image `amos2.dmp`:

```
/* Assume we start with system image amos2.dmp */
create function sqrt(real x) -> real as foreign ’sqrtbf’;
             /* Add foreign function sqrt to datamase image */
save "mydb.dmp"; /* Make own extended image */
quit;
```

We can now create the empty database image by issuing this command:

```
myApp amos2.dmp init.amosql
```

Then we can call our application with

```
myApp mydb.dmp
```

3.1.6 Trapping errors in foreign functions

In the previous examples there is no explicit memory application and the functions will work correctly with catcherror always set to FALSE. If an error happened during any of the Amos II system calls the system will first print the error message and then throw the control through the foreign function to an error trap outside it.

However, often you need to do some finalization before exiting the foreign function. In that case you must always trap the Amos II system calls from within the foreign function, do the finalization, and then return the control to Amos II in a clean way. The system provides an error trapping facility, called *unwind-protection*, that provides a simple way to trap all Amos II errors and always execute clean-up code before exiting a C-block. To unwind-protect a piece of C-code use the following code skeleton:

```
{unwind_protect_begin;
       /* Place code to be protected here */
unwind_protect_catch;
       /* This code will always be executed. If an Amos II error happened in the protected
              code the system will print the error message, then trap the error and pass
              the control here */
unwind_protect_end;}
/* The control comes here ONLY if the execution of the protected code was successful */
```

For example, the following function, `ceval(Charstring s)-> Vector`, illustrates how to trap errors in foreign functions. It takes an arbitrary long string containing an AmosQL statement as input, calls the system to evaluate the statement, and then emits the result tuples from the evaluation.

```
void ceval(a_callcontext cxt, a_tuple argl)
{
       char *stmt=NULL; /* Will contain AmosQL statement */
       int stmt_size;
       dcl_scan(s);
       dcl_tuple(tpl);

       {unwind_protect_begin; /* We just trap everything for simple deallocation */
              stmt_size = a_getelemsize(tpl,0,s,FALSE);
              stmt = malloc(stmt_size+1); /* One extra for the trailing NULL character */
              if(stmt!=NULL) /* malloc may actually fail if memory exhausted */
              {
                     a_getstringelem(tpl,0,stmt,size,FALSE);
                     a_execute(cxt->connection,s,tpl,stmt,FALSE);
                     while(!eof(s))
                     {
                            a_setobjectelem(argl,1,tpl,FALSE);
                            a_nextrow(s,FALSE);
                     }
              }
       unwind_protect_catch;
              if(stmt!=NULL) free(stmt);
                     /* stmt will be NULL if there was a failure before a_getstringelem */
              /* You must always deallocate the local handles: */
              a_closescan(c,s,TRUE); /* This is done automatically by free_scan(s) too */
              free_scan(s);
              free_tuple(tpl); /
       unwind_protect_end;}
       return; /* Control comes here ONLY when the entire call was successful */
}
```

The alternative to using unwind-protection is to trap every individual system calls and ’manually’ print the eventual error messages. It should then be noted that `a_emit` may fail without setting the error indication in the case when the caller does not need any more result tuples or when there are error exceptions raised by some foreign function receiving the emitted tuple. To be able to trap specifically the case of no more result tuples needed by the consumer(s), there is a flag `cxt->done` which is set to TRUE whenever the consumer(s) does not need any more tuples.

3.1.7 Raising errors exceptions

The following system function can be used to raise an exception in a foreign function:

```
int a_error(int errno, int obj, int catcherror)
```

It will signal system error `errno` for the object `obj`. If `catcherror` is FALSE the system will do an internal error jump and the control will not return to the caller. If `catcherror` is TRUE the error exception will be raised after the foreign function is exited. The possible error codes correspond to the code in `a_errno` and are defined as C macros in `storage.h`. With each error code there is an associated error text string. The result is TRUE if the exception was successfully raised.

In case you want to introduce your own error codes, you can define a new error code for an error text with the system function

```
int a_register_error(char *text)
```

It assigns a unique error code to the specified error text. The error code is returned. If `a_register_error` is called twice with the same error text, the same error code will be returned.

3.1.8 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 sqrt:

```
sqrt(4);    /* 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 instead of some 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 [2]. 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.

Simple foreign Amos II functions is a special case where all arguments are known and all results are unknown. For example the binding pattern of sqrt is the list "bf".

The square root 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 sqrt above.

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

However, case (2) cannot be handled by the simple foreign Amos II function sqrtbf as it requires the computation of the square rather than the square root. Case (2) thus requires a multi-directional foreign function and we will now spec-

ify a generalized sqrt using that mechanism.

3.1.8.1 Binding multi-directional foreign functions

To implement a multi-directional foreign function you first need to think of for which binding patterns implementations are needed. In the sqrt 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 C function sqrtbf in Sec. 3.1.1 implements the "bf" case. The "fb" case is implemented by the following C function, sqrtfb:

```
void sqrtfb(a_callcontext cxt, a_tuple tpl)
/* sqrtbf implements the inverse of the square root, i.e. the square */
{
       double r;

       r = a_getdoubleelem(tpl,1,FALSE); /* Pick up the result */
       a_setdoubleelem(tpl,0,r*r,FALSE); /* The argument x = r^2 */
       a_emit(cxt,tpl,FALSE); /* Only one result */
       return;
}
```

A multidirectional foreign function implementation is bound to a symbol just as simple foreign functions. For example:

```
       a_extfunction("sqrtfb",sqrtfb);
```

3.1.8.2 Creating the resolvent for a multi-directional foreign function

See [1] for documentation of how to define multidirectional foreign functions. The multi-directional foreign function is created by executing the AmosQL statement:

```
create function <fn>(<argument declarations>) -> <result declaration>
       as multidirectional (’<bpat> foreign '<name>')...;
```

For example:

```
create function sqrt(real x) -> real as multidirectional
              (’bf’ foreign 'sqrtbf')(’fb’ foreign ’sqrtfb’);
```

As for simple foreign function definitions, <fn> is the name of the foreign function, <argument declarations> is the signature of its arguments, and <result signature> is the signature of its results. In addition for each binding pattern <bpat> a corresponding foreign function name <name> is specified.

Separate cost hints can be assigned to each binding pattern [1].

3.1.8.3 The driver program

The complete driver program for the multidirectional foreign function sqrt would look like this:

```
#include "callout.h"
main(int argc,char **argv)
{
       dcl_connection(c); /* To hold connection to Amos */

       init_amos(argc,argv); /* Initialize embedded Amos */
       a_connect(c,"",FALSE); /* Connect to embedded Amos */

       /* Bind C function 'sqrtbf' to Amos symbol 'sqrtbf': */
       a_extfunction("sqrtbf",sqrtbf);
       a_extfunction("sqrtfb",sqrtfb);
```

```
      /* Define Amos II function 'sqrt(real x) -> real' as multidirectional
            (can also be done in the Amos top loop): */
      a_execute(c,s,"create function sqrt(real x) -> real \
            as multidirectional ('bf' foreign 'sqrtbf')('fb' foreign 'sqrtbf');",
            FALSE);
      free_connection(c);
      amos_toploop("Amos"); /* Enter interactive Amos II top loop */
      exit(0);
}
```

3.2 Implementing Amos II functions in ALisp

A foreign function implementation in ALisp has the structure

```
(defun <fn> (fno <arg1>... <res1>...) . <body>)
```

For example:

```
(defun sqrtbf (fno x r)
      (cond ((= x 0) (osql-result 0.0 0.0))
            ((> x 0)
             (let ((r (sqrt x)))
                   (osql-result x r)
                   (osql-result x (minus r))
             )))
```

The first argument fno of a Lisp implementation of a multi-directional foreign function is bound by the system to the Amos II function object. It is usually not used only for printing error messages. The other arguments correspond to the arguments and results of the Amos II function. <arg1>... are the arguments, and <res1>... are the results. The bound values of the arguments or results are bound to the corresponding Lisp variable. (The unbound arguments or results are bound to the atom *). Finally, <body> is the Lisp function body.

The *result tuple* tpl is emitted by calling

```
(osql-result &rest tpl)
```

for example

```
(osql-result x r)
```

`osql-result` corresponds to the `a_emit` in the C callout interface. The number of arguments of `osql-result` (length of tpl) must match the sum of the arity and the width of the Amos II function.

As in the sqrt example, `osql-result` can be called several times for functions returning a stream of result tuples.

Function *binding* is not needed for foreign functions implemented in ALisp.

A simple foreign function in ALisp is *created* just as simple foreign functions i C, for example:

```
create function sqrt(real x) -> real as foreign 'sqrtbf';
```

No *driver program* is needed for foreign Amos II functions implemented in ALisp.

We can now use sqrt in AmosQL statements:

```
amos 1> sqrt(4.0);
        2.0
        -2.0
```

The following expression defines the Lisp function sqrt2 that returns a tuple of the two square roots of a number:

```
(defun sqrt2 (fno x r1 r2)
        (cond ((>= x 0.)
               (setq r (sqrt x)))
               (osql-result r (minus r))))
```

Definition of sqrt2 in AmosQL:

```
create function sqrt2(number x) -> <number r1, number r2> as foreign 'sqrt2';
```

Example of sqrt2 call in AmosQL:

```
AMOS 1> sqrt2(4.0);
        <2.0,-2.0>
```

3.2.1 Multi-Directional Foreign Functions in ALisp

For each binding pattern of a multi-directional foreign function you must implement separate Lisp functions. In the sqrt case these Lisp functions are:

```
(defun sqrtbf (fno x r) ; square root
      (cond ((= x 0) (osql-result x 0.0))
            ((> x 0)
             (setq r (sqrt x))
             (osql-result x r)
             (osql-result x (minus r)))))

(defun sqrtfb (obj x r) ; square
      (osql-result (* r r) r))
```

In sqrtbf the argument x is bound to a number and r is to be computed. In sqrtfb the argument r is bound to a number and x is to be computed. Multi-directional foreign functions in Lisp are created in AmosQL just like multi-directional foreign functions in C, for example:

```
create function sqrt(real x) -> real as multidirectional
       ('bf' foreign 'sqrtbf')('fb' foreign 'sqrtfb');
```

We are now ready to test our multi-directional foreign function definition of sqrt:

```
AMOS > sqrt(4);  /* Two results */
       2.0
       -2.0
AMOS > sqrt(0);  /* One result */
       0.0
AMOS > sqrt(-1); /* No result */
AMOS > select x for each number x where sqrt(x)=4.0;
       16.0
AMOS > select true where sqrt(2)=4;
AMOS > select true where sqrt(4)=2;
       TRUE
```

3.2.2 Aggregation Operators

Aggregation operators are Amos II functions that take bags of values as arguments. Aggregation operators are defined using foreign functions in Lisp calling a special Lisp function, `mapbag`, to iterate over the elements of a bag:

```
(mapbag <bag> <mapfn>)
e.g.
(defun mycountbf (fno x r) ; count number of elements in a bag
       (let ((cnt 0))
              (mapbag x #'(lambda (row)(setq cnt (add1 cnt))))
              (osql-result x cnt)))
(osql "create function mycount(bag x)->integer as foreign 'mycountbf';")
```

`mapbag` iterates over all elements in a bag and applies `mapfn` on each of them. `mapfn` has a single argument, row, which will be bound to a list representing the elements of the bag. Thus not only bags of single values, but also bags of tuples of values are supported.

For example:

```
(defun mysumbf (fno x r) ; sum of numbers in a bag
       (let ((sum 0))
              (mapbag x #'(lambda (key)(setq sum (+ sum (car key)))))
              (osql-result x r)))
(osql "create function mysum(bag of number x)-> number r as foreign 'mysumbf';")
```

3.2.3 Terminating foreign Amos II functions

Iterations over bags sometimes need to be terminated before the entire bag is iterated over. The CommonLisp functions catch and throw [3] can be used for terminating iterations.

For example, the foreign Amos II function `some(Bag b)->Boolean` returns TRUE if there are no elements in a bag, i.e. *some* actually implements existential quantification of sub-queries. It is implement as an iteration over the bag using `mapbag`, but where the iteration is terminated after the first element found. If a single element was found the value TRUE is returned.

```
(defun someb (fno x r)
       (catch 'some
              (mapbag x #'(lambda (key)
                     (osql-result x 'true)
                     (throw 'some nil)))))
(osql "create function some (bag of object x) -> boolean as foreign 'someb';")
```

WARNING: Only catch and throw can be used if an iteration is to be terminated. Do not use other Lisp functions, such as return, or unwind-protect to terminate iterations as they will cause problems.

The system Amos II function `notany` can be implemented as

```
(defun notanyb (fno x r)
       (if (not (catch 'notany (mapbag x (function (lambda (x)(throw 'mapbag t))))))
              (osql-result x 'true))))
(osql "create function notany(bag of object x) -> boolean as foreign 'notanyb';")
```

References

1 Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld, *Amos II Release 7 User's Manual*, http://www.it.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: *ALisp User's Guide*, UDBL Technical Report, Dept. of Information Technology, Uppsala University, Sweden http://user.it.uu.se/~udbl/amos/doc/alisp.pdf

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

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