aStorage

a main-memory storage manager

Tore Risch
Uppsala Database Laboratory
Department of Information Technology
Uppsala University
Sweden

Tore.Risch@it.uu.se

2009-09-03

The *Amos II* DBMS uses a main memory database storage manager called *aStorage*. All data in an *Amos II* database is stored in a *database image* managed by *aStorage*. The storage manager is scalable allowing data structures to grow very large gracefully and dynamically without performance degradation. The system includes a garbage collector that is incremental and based on reference counting techniques. This means that the system never needs to stop for storage reorganization and makes the behaviour of the system very predictable. The storage manager is extensible so that users can define new kinds of object, *storage types*, managed by the system. The system is written in ANSII C. *aStorage* is tightly integrated with a Lisp system called *aLisp*. New *aLisp* data types can be defined in C and made interoperable between *aLisp* and C.

This report documents *aStorage*. It also explains how to extend *aLisp* with new datatypes and functions.

Table of contents

1. Introduction

aStorage is a main memory storage manager that represents data in the *Amos II* DBMS [3]. *aStorage* is responsible for allocation and deallocation of physical objects inside the database image. A physical object is a C structure handled by *aStorage*. All data in an Amos II database are internally represented as physical objects managed by *aStorage*. The C/C++ programmer can define own persistent data structures as physical objects by using a set of storage manager primitives.

The types and functions defined by the *AmosQL* language [1] are on a higher level than the basic data primitives managed by *aStorage*. The storage manager handles physical objects in the database image called *storage types,* while the *Amos II* kernel handles high level *logical types* defined by the *AmosQL* language. This document describes the interface to the storage types only.

The storage manager is independent of the rest of the *Amos II* system. *Amos II* has several system layers on top of the storage manager. An important layer is a Lisp interpreter, *aLisp* [2], which is tightly interfaced with *aStorage*. A large part of *Amos II* is written in *aLisp*. The *aLisp* system is documented separately. This document includes a description of how to extend *aLisp* with new data types and functions written in C.

With *aStorage* the C implementer has the choice of allocating data *persistently* by using a set of primitives provided by the storage manager. Persistency in this case means that data is allocated inside a memory area called the *database image,* which can be saved on disk and later restored. Persistent data is saved to disk when the user calls the C function *a_rollout(char *filename),* issues the *AmosQL* statement *save*; [1], or call *aLisp*'s *ROLLOUT* function [2]. The image is restored when restarting the system with the image file as command line argument.

Another important service of the storage manager is to provide a garbage collection subsystem that automatically deallocates persistent memory no longer in use in the database.

Data can also be allocated *transiently* by using the usual C routines *malloc*, etc., but transient data cannot be saved on disk and are lost when the system exits. Furthermore the programmer is responsible for deallocating transient data manually, as C has no automatic garbage collector.

1.1. Handles

All access to physical objects is made through *handles* which are indirect identifiers for physical data records in C. The representation of handles is unsigned integers. In order to make the application code both fast and independent of the internal representation of handles, the handles are always manipulated through a set of C macros and utility functions. The interface with the

storage manager is defined by the header file `storage.h`. The interface is connected to an automatic garbage collector so that data no longer used is reclaimed when using those macros/functions.

Handles to persistent objects must be declared of C type `oidtype` and *must* always be initialized to the global C constant `nil` using the C macro `dcl_oid(x)`. For example:

```
dcl_oid(myhandle);
```

1.2. Physical Objects

With every handle there is an associated C structure, the *physical object*, stored in the database image and holding the *value* of the handle. The physical data objects are C structures containing the data stored persistently in the database image together with a *physical* *`type`* identifier identifying the type of the object. The physical objects are accessed indirectly through the handles. The layout of the physical data object depends on the data type. However, the first two bytes of a physical object are *always* reserved for the system; the succeeding bytes are used for storing the data. Every persistent data item must be represented as physical objects, including literals such as integers and strings. For example, integers are represented by this structure:

```
struct integercell
{
  objtags tags;
  short int filler;
  int integer;
};
```

The field `tags` is used by the system, the field `integer` stores the actual integer value, and `filler` aligns the integer to a full-word.

The header of a physical object (field `tags` with type `objtags`) is maintained by the storage manager. It contains the identification of its physical type (1 byte) and a *reference counter* (1 byte) used by the automatic garbage collector.

Every physical type has an associated *type identifier* number and a unique *type name* string known to the storage manager. The main memory array `typefns` represents information about storage types. Since the type identifier is represented by one byte there can be up to 256 physical types defined. A number of physical storage types are predefined, including LIST, SYMBOL, INTEGER, REAL, EXTFN (an *aLisp* function defined in C), CLOSURE (internal *aLisp* closures), STRING, ARRAY (1D fixed size arrays), STREAM (file streams), TEXTSTREAM (streams to text buffers), HASHTAB (hash tables), ADJARRAY (dynamically extensible 1D arrays), and BINARY (bit strings). In `storage.h` there are structure definitions defined for the physical representation of most of the built-in storage types. The convention is used that if the type is named *`xxx`* the template has the name *`xxx`*`cell`, e.g. REAL has a template named `realcell`, etc. The type identification numbers for most built-in types are defined as C macros

in `storage.h`, with the convention that a type named *xxx* has a corresponding identification number *XXX*`TYPE` if it is defined as a C macro or *xxx*`type` if it is bound to a global C variable. For example, physical objects representing integers are identified by the data type tag `INTEGERTYPE` stored as the 2nd byte in field `tags` of `integercell`.

The C/C++ programmer can extend the built-in set of physical data types with new persistent data structures through the C function `a_definetype`, explained below. It defines to the storage manager the properties of the new data type.

1.3. Logical Data Objects

Notice the difference between *physical* and *logical objects*: Physical data objects are C record structures stored in the database image while logical data objects are object descriptors referenced in AmosQL. Logical data objects are internally represented by one or several physical data objects. For example, Amos II objects of logical data type INTEGER are directly represented by the above mentioned physical data objects also named INTEGER. Similarly, other simple literal objects (e.g. real numbers and strings) are internally represented as directly corresponding physical objects. More complex objects, e.g. the logical datatype FUNCTION, are represented by data structures consisting of several physical objects of different types. *Surrogate objects* in AmosQL are represented as physical objects of a particular kind named `OID` with type tag `SURROGATETYPE` describing properties of the logical object identifier of the surrogate. One property of an OID object is a numeric identifier maintained by the storage manager; another one is a handle referencing the Amos II type(s) for the logical object. References to OID objects are very common in the database, e.g. to represent arguments or values of functions, extents of types, etc.

The AmosQL user cannot directly manipulate physical objects; they can only be manipulated in C/C++ or *aLisp*.

1.4. Dereferencing

In order to access or change the contents a physical object given a handle it has to be converted from a handle into a C pointer to the corresponding physical object in the database image. This process is called to *dereference* the handle. The dereferencing of physical data objects is very fast and does not involve any data copying; it involves just an offset computation.

Once the physical object has been dereferenced its contents can be investigated by system provided C macros and functions or directly by C pointer operations. However, **notice** that the data in the image may move when new data is allocated, so the programmer must only keep pointers to physical objects when it is guaranteed that no new data is allocated in the image. To be safe physical objects should always be accessed by dereferencing handles.

The following C macro dereferences a handle:

```
dr(x,str)
```

`dr` returns the address of the record referenced by the handle `x` casted as a C struct named `str`. For example, if the C variable `ic` contains a handle to an integer, the actual integer's value is accessed with `dr(ic,integercell)->integer`.

The following C function prints an integer referenced by the handle `ix`:

```
void printint(oidtype ix)
{
   struct integercell *dix = dr(ix, integercell);

   printf("ix = %d\n", dix);
}
```

Notice that here the parameter `ix` must be a handle referencing an object of type INTEGER, otherwise the system might crash. To make `printint` safe it therefore should check that `ix` actually references an integer. The following C macro can be used for investigating the type of a physical object handle:

```
a_datatype(x)
```

`a_datatype` returns the type identifier of a handle x.

For example, the function `printint2` checks that `ix` actually is an integer before printing its value:

```
void printint2(oidtype ix)
{
   if(a_datatype(ix) == INTEGERTYPE)
      printf("ix = %d\n",dr(ix,integercell)->integer);
   else printf("ix is not an integer\n");
}
```

WARNING: Storage manager operations may invalidate dereferenced C pointers because the derefrenced objects might move to other memory locations when the image is expanded. Thus dereferenced pointers may become incorrect once a system feature that causes the image to expand is called. Object allocation is the only system operation that may cause this. Thus, if a system function is called that is suspected to do object allocation (most do), the dereferencing *must* be redone. It is therefore safer to always dereference through `dr` as in `printint2` rather than saving a dereferenced C pointer as in `printint`.

1.5. Assigning handles to locations

In order for the storage manager and garbage collector to function correctly, C locations (variables or fields) of type `oidtype` *must* be initialized to the global variable `nil` by the C

macro `dcl_oid(x)`. To update the location the following C macro *must* be used:

```
a_setf(location,value);
```

`a_setf()` corresponds to an assignment, `location=value`, but, unlike an assignment, it also updates the reference counter of `value` so it is increased after the assignment. The reference counter increment indicates to the system that the C field `location` holds a reference to the physical object and it therefore cannot be deallocated until the handle location is *released*, meaning that the location does not need to access the object any more. A handle location `x` is released with the C macro:

```
a_free(x)
```

The object x will not be physically removed from the database image if there is some other location still holding a reference to it. `a_free(x)` is equivalent to `a_setf(x,nil)` but faster.

No other location holds a reference to a physical object if the reference counter is 0. Thus, when the reference counted is decreased to 0 by `a_setf()`, the physical object is passed to the garbage collector for deallocation from the image. Thus, unlike the C function *free()*, `a_free()` will deallocate `x` *only* when there is no other location holding a reference to it.

To handle reassignments of locations correctly, `a_setf()` decreases the reference count of the handle *previously* referenced from `loction` and increases the reference counter of `value`.

Lisp symbols (e.g. `nil`) are not garbage collected and thus not reference counted.

Notice that the location *must* be assigned to some handle before `a_setf()` can be used, otherwise the system is likely to crash when trying to release a non-existing handle. It is therefore required to *always* initialize C handle locations using `dcl_oid()` when declaring them. An alternative is to use the macro `a_let()` the first time a location is assigned a handle. It assumes the old value of `loction` was uninitialized and will therefore only increase the reference counter of `value`, while ignoring the old value in `location`:

```
a_let(location,value)
```

1.6. Allocating physical objects.

Physical objects inside the database image can be allocated only through a number of storage manager primitives (not through e.g. *malloc()*). When a physical object is allocated it initializes the reference counter to 0. The built-in datatypes have allocation macros and functions defined in `storage.h`, e.g.:

`mkinteger(xx)`	allocates a new integer object.
`mkreal(xx)`	allocates a new double precision real number object.
`mkstring(xx)`	allocates a new string object.

`mksymbol(xx)`	allocates or gets the symbol named *capitalized* `xx`.
`cons(x,y)`	allocates a new list cell.

For example, the following C function adds two integers:

```
oidtype add(oidtype x, oidtype y)
{
   int sum;

   if(a_datatype(x) != INTEGERTAG ||
      a_datatype(y) != INTEGERTAG)
   {
      printf("Cannot add non-integers\n");
      exit(1);                    ← Could call error manager here.
   }
   sum = dr(x,integercell)->integer + dr(y,integercell)->integer;
   return mkinteger(sum);
}
```

The following code fragment allocates two integers, calls `add()`, and prints the sum.

```
{
   dcl_oid(x), dcl_oid(y), dcl_oid(s); // Local handles must be initialized!

   a_setf(x,mkinteger(1)); // assign x to new integer 1
   a_setf(y,mkinteger(2)); // assign y to new integer 2
   a_setf(s,add(x,y));     // assign s to new integer being sum of a x and y
   printf("The sum is %d\n",dr(s,integercell)->integer);
   a_free(s);              // release locations s, x, y
   a_free(x);
   a_free(y);
}
```

In `storage.h`, for each built-in storage type there is a C constant (upper case) or a variable (lower case) containing the identifier for the type.

Type-name	constant/variable	Short description
`LIST`	`LISTTYPE`	Lists
`SYMBOL`	`SYMBOLTYPE`	Symbols
`INTEGER`	`INTEGERTYPE`	Integers
`REAL`	`REALTYPE`	Double precision reals
`EXTFN`	`EXTFNTYPE`	aLisp function in C
`CLOSURE`	`CLOSURETYPE`	aLisp function closure
`STRING`	`STRINGTYPE`	Strings
`ARRAY`	`ARRAYTYPE`	1D Arrays
`STREAM`	`STREAMTYPE`	File streams
`TEXTSTREAM`	`TEXTSTREAMTYPE`	String streams
`SOCKET`	`sockettype`	Socket streams
`HASHTAB`	`HASHTYPE`	Hash tables
`HASHBUCKET`	`HASHBUCKETTYPE`	Internal to hash tables
`OID`	`SURROGATETYPE`	Object identifiers for surrogate objects
`HISTEVENT`	`histeventtype`	Update events

For most built-in datatypes there are C macros or functions for construction and access. For example, to allocate a new handle of type STRING with the content “Hello world” you can use the macro `mkstring()` that returns a handle to the new string:

```
{
   dcl_oid(mystring);
   ...
   a_setf(mystring,mkstring("Hello world"))
   ...
   a_free(mystring);
};
```

To dereference a handle referencing a STRING object the macro `getstring` can be used:

```
{
   dcl_oid(mystring);
   char *mystringcont;

   a_setf(mystring,mkstring("Hello world"));
   mystringcont = getstring(mystring);
   printf("%s\n",mystringcont);
   a_free(mystring);
};
```

The following are examples of C library functions and macros used for manipulating the built-in data types:

```
oidtype mkinteger(int x)          (macro) Construct handle for a new integer
int integerp(oidtype x)           (macro) TRUE if X is a handle for an integer
int getinteger(oidtype x)         (macro) Dereference a handle for an integer

oidtype mkreal(double x)          (macro) Construct handle for a new real
int realp(oidtype x)              (macro) TRUE if X is a handle for a real
double getreal(oidtype x)         Dereference a handle for a real

oidtype mkstring(char *x)         (macro) Create handle for a new string
int stringp(oidtype x)            (macro) TRUE if X is a handle for a string
char *getstring(oidtype x)        (macro) Dereference a handle for a string

oidtype new_array(int size,oidtype init)
                                  Construct handle for a new array with elements init
int arrayp(oidtype x)             TRUE if X is a handle for an array
int a_arraysize(oidtype arr)      return the array size
oidtype a_seta(oidtype arr,int pos,oidtype val)
                                  Set an array element
oidtype a_elt(oidtype arr,int pos)
                                  Retrieve array element
oidtype a_vector(oidtype x1,...,xn,NULL)
                                  Create a new array and its elements x1 ... xn.

oidtype cons(oidtype x,oidtype y) Create handle for a new list cell
int listp(oidtype x)              (macro) TRUE if X is a list cell
oidtype hd(oidtype x)             (macro) Head of list cell
oidtype tl(oidtype x)             (macro) Tail of list cell
oidtype a_list(oidtype x1,...,xn,NULL)
```

	Create new list of `x1 ... xn`
`oidtype mksymbol(char *x)`	(macro) Create a new symbol
`int symbolp(oidtype x)`	(macro) `TRUE` if `X` symbol
`oidtype globval(oidtype x)`	(macro) Get global value of symbol.
`char *getpname(oidtype x)`	(macro) Get print name of symbol
`a_print(oidtype x)`	Print object of any type. Very useful for debugging.
`oidtype t`	Symbol `T` representing TRUE
`oidtype nil`	Symbol `NIL` representing empty list and FALSE

1.7. Defining storage types

This subsection describes how to introduce new physical storage types to *aStorage*. This is required when new C data structures need to be defined for *aLisp* or Amos II.

In `storage.h` the basic built-in physical storage type tags are declared as macros. The include file also contains the record templates for each storage type.

There is a global *type table* which associates a number of optional C functions with each physical object type. A new storage type is introduced into the system (thus expanding the type table) with a call to the C function `a_definetype()`:

```
int a_definetype(char *name,
          void (*dealloc_function) (oidtype),
          void (*print_function) (oidtype,oidtype,int))
```

`a_definetype()` adds a new type named `name` to the type table and returns the new type identifier as an integer.

`dealloc_function()` is a required C function taking an object of the new type as argument. It is a *destructor* called only by the garbage collector when the object is deallocated. It shall then release all locations referenced by the object and call storage manager primitives to deallocate the storage occupied by the object.

`print_function()` is an optional *print function* called by PRINT to provide a customized printing of physical objects of the new type. See section 1.8.1.

1.8. Streams

aLisp has several data types representing streams:

`STREAM`	represents regular C file streams.
`TEXTSTREAM`	represents streams over buffers in the database image.
`SOCKET`	represents socket streams for communication with other aLisp systems.

The following system standard streams are defined:

`oidtype stdinstream` for C's standard input stream

`oidtype stdoutstream` for C's standard output stream

`oidtype stderrstream` for C's standard error stream

Streams are physically represented as other data types but with some special stream attributes in the beginning of the structure template:

```
struct xxxcell
{
  objtags tags;
  short int bytes;             /* Total size of object in bytes, incl. header */
  char autoflush;                       /* Flush after each item and new line */
  char filler[3];                                          /* Unused flags */
  int line_num;                                      /* Current line number */
  oidtype logstream;                    /* Stream to copy input to if non-NIL */
  /*** end of stream header ***/
```

The attributes above must always be present for stream templates. Additional specific attributes can be added after the end of the stream header. Once the data type has been defined using `definetype()` the newly created type can be made into a stream by a call to a `define_stream()` implementation:

```
int a_define_stream_implementation(int tag, /* Storage type */
                                int(*getc)(oidtype),
                               int(*ungetc)(int,oidtype),
                               int(*feof)(oidtype),
                               int(*puts)(char*,oidtype),
                               int(*putc)(int,oidtype),
                               int(*fflush)(oidtype),
                               int(*fclose)(oidtype));
```

The first argument, `tag`, is the type tag (returned by `definetype()`) of the type to be made a stream. Each stream should have the following associated functions (methods):

`int getc(oidtype stream)` Returns the next character in stream.

`int ungetc(int c, oidtype stream)`
Put back character c in stream.

`int feof(oidtype stream)` Return TRUE if end-of-file reached.

`int putc(int c, oidtype stream)`
Write character c to the stream

`Int readbytes(oidtype stream, void *block, unsigned int len)`
Read a block of data from the stream. The slower `putc` method is used if this method is NULL.

`int writebytes(oidtype stream, void *block, unsigned int len)`
Write a block of data to the stream. The slower `getc` method is used if this method is NULL.

`int fflush(oidtype stream)` Flush stream buffer contents.

`int fclose(oidtye stream)` Close the stream.

Once these methods are defined and registered the user can use the following generic stream functions to manipulate the new stream:

`int a_getc(oidtype stream);`	Read one character
`int a_ungetc(int c, oidtype stream);`	Unread one character
`int a_puts(char *str,oidtype stream);`	Write string
`int a_writebytes(oidtype stream, void *buff, unsigned int len);`	Write block
`int a_putc(int c, oidtype stream);`	Write a character
`int a_readbytes(oidtype stream, void *buff, unsigned int len);`	Read block
`int a_fclose(oidtype stream);`	Close stream
`int a_feof(oidtype stream);`	Test for end-of-file
`int a_fflush(oidtype stream);`	Flush stream buffer

The performance of stream management can be improved by moving bulks of data to or from the stream through calls to `a_printbytes()` and `a_readbytes()`. If the corresponding methods are not registered with a stream, writing to and reading from the stream is slower.

1.8.1. Marshalling objects

Streams are often used for writing object in such a format that they can later be restored by reading. This is particularly important when using streams to communicate data between *aLisp* peers, e.g. using sockets [2]. The Lisp function PRINT prints object structures on a stream in such a format (S-expression) that copies of the objects are later be allocated when the function READ is reads the object from the stream. This PRINT and READ are Lisp's generic (de-)marshalling functions. Lisp's S-expression notation provides standardized marshalling and demarshalling for the basic Lisp datatypes. In addition customized (de-)marshalling can be specified for user defined storage type, as will be described below.

In C the following functions can be used for (de-)marshalling S-expressions:

`oidtype a_read(oidtype stream)`	Read (unmarshal) S-expression from a stream. This corresponds to the Lisp function READ.
`oidtype a_print(oidtype s)`	Print S-expression a followed by a line feed on `stdoutstream`, normally for debugging.
`oidtype a_printobj(oidtype s, oidtype stream)`	Print S-expression s followed by a line feed as delimiter on stream. This corresponds to the Lisp function PRINT.
`oidtype a_prin1(oidtype s, oidtype stream, int princflg)`	Print S-expression s on stream. If `princflg` is FALSE the printout be marshalled for subsequent reading; if `princflg` is TRUE object will be written as PRINC and cannot be read using `a_read`. Notice that, since no delimiter is inserted as with `a_printobj()`, it is up to the user to ensure proper object delimitation.

`oidtype a_terpri(oidtype stream)`

Write a line feed on the stream.

2. Interfacing Lisp with C

An *aLisp* function can be implemented as a C function and C functions can call *aLisp* functions. *aLisp* and C can also share data structures without data copying or transformations. The error management in *aLisp* can be utilized in C as well for uniform and efficient error management.

In order to interface *aLisp* with C/C++ you must include the file `alisp.h` in your C program. In the development version, the file `democpp.cpp` contains a simple C program that calls *aLisp* and where *aLisp* also calls C.

This section describes how to call C functions from *aLisp*, and how to call *aLisp* functions from C.

2.1. Calling C from Lisp

As a very simple example of an external Lisp function we define an *aLisp* function HELLO which prints the string 'Hello world' on the standard output. It has the C implementation:

```
#include "alisp.h"
oidtype hellofn(bindtype env)
{
   printf("Hello world\n");
   return nil;
}
```

The include file `alisp.h` contains all necessary declarations for implementing external Lisp functions in C; External Lisp function definitions must always return handles of type `oidtype`. Do not forget the `return` statement, otherwise the system might crash!

In order to be called from Lisp, an external Lisp function implementation has to be registered with a symbolic *aLisp* name, in this case the symbol HELLO, by calling:

```
extfunction0("HELLO",hellofn);
```

A system convention is that an external Lisp function named `XXX` is named `xxxfn` in C, as for `HELLO`.

The call to register an external Lisp function should be done in a main C program, the *driver program*, after the system has been initialized (i.e. after `init_amos()` or `a_initialize()`

is called). The following driver program initializes the system, registers HELLO, and calls the *aLisp* read-eval-print loop with prompter string 'Lisp>'.

```
#include "alisp.h"

oidtype hellofn(bindtype env)
{
   printf("Hello world\n");
   return nil;
}

void main(int argc, char **argc)
{
   init_amos(argc,argv);
   extfunction0("HELLO",hellofn);
   evalloop("Lisp>");
}
```

When the above program is run the user can call HELLO from the read-eval-print loop by typing

```
(hello)
```

2.1.1. Defining external Lisp functions in C

Lisp functions can be implemented as *external Lisp functions* in C. An external *aLisp* function `fn()` with optional arguments `x1, x2,..., xn` must have the following signature in C:

```
oidtype fn(bindtype env,oidtype x1,oidtype x2,..,oidtype xn)
```

The first argument `env` is the *binding environment* to be used by the system for error handling, memory management, and other things.

For example, the following function implements an *aLisp* function to add two numbers:

```
oidtype addfn(bindtype env, oidtype x, oidtype y)
{
   int ix, iy, r; // will hold integer values of x, y and result

   IntoInteger(x,ix,env); // Retrieve value of integer x into ix and raises
                          // aLisp error if x is not an integer object
   IntoInteger(y,iy,env); // This will not be executed if x is not an integer
   r = ix + iy;           // Both x and y must be integers for this to execute
   return mkinteger(r);   // Return a new physical integer object
}
```

`addfn` is registered with

```
exfunction2("add",addfn);
```

The number '2' after '`extfunction`' indicates that this *aLisp* function takes two arguments.

External Lisp functions need to be very careful to check the legality of the handles they receive,

otherwise the system may crash. To check that a handle is of an expected type use the C macro:

```
OfType(x,tpe,env)
```

A standard error will be generated if `x` does not have the type tag `tpe`. For integers the above used macro `IntoInteger()` is a convenient alternative to `OfType`.

External Lisp functions are *registered* (assigned to *aLisp* symbols) by calling a system C function:

```
extfunctionX(char *name, Cfunction fn);
```

`name` is the *aLisp* name for the external Lisp function

`fn` is the address of the C function.

Different versions of `extfunctionX()` are available depending on the arity X of the external Lisp function. For example,

```
extfunction2("add",addfn);
```

There are corresponding *aLisp* registration functions for functions with arity 0, 1, 2, 3, 4, 5 named `extfunction0()`, `extfunction1()`, etc.

When a physical object handle whose reference counter has been managed by `a_setf()` is to be returned from a C-function the following C-macro should be used:

```
a_return(x);
```

`a_return()` returns `x` from the C-function after the reference counter of `value` has been decreased *without* deallocating `x` if the counter reaches 0.

For example, the following external Lisp function calls `addfn()` twice to sum three integers:

```
oidtype add3fn(bindtype env, oidtype x, oidtype y, oidtype z)
{
   dcl_oid(s);

   a_setf(s,addfn(env,x,y));
   a_setf(s,addfn(env,s,z));
   a_return(s);
}
```

The variable `s` holds the result from `add3fn()`. If it had been returned by the C statement

```
   return s;
```

the result object would never be released from the location `s` since the reference counted would not have been decreases, and there would be a memory leak.

For example, the following function reverses a list:

```
oidtype reversefn(bindtype env, oidtype l)
{
   ccl_oid(lst), dcl_oid(res);

   a_setf(lst,l);
   while(listp(lst))
   {
      a_setf(res,cons(hd(lst),res));
      a_setf(lst,tl(lst));
   }
         a_free(lst);
   a_return(res);
}
```

Register REVERSE with:

```
extfunction1("REVERSE",reversefn);
```

WARNING: You *cannot* assign C function parameters (such as `l` in the example) with `a_setf()` or release them with `a_free()`. C function parameters are not reference counted. Instead the parameter `l` is assigned to the local variable `lst` in order to subsequently use `a_setf()`. C function parameters are returned using `a_return()`.

WARNING: The C implementation of an external Lisp function must always return a legal handle, otherwise the system might crash. It is therefore recommended to run the system in 'debug mode' while testing external Lisp function where the system always checks the legality of data passed between *aLisp* from C.

2.1.2. Variable arity external Lisp functions

Variable arity external functions accept any number of arguments. External Lisp functions with more than 5 arguments also need to be defined as variable arity functions. Variable arity external Lisp functions have the signature:

```
oidtype fn(bindtype args,bindtype env)
```

where `env` is the binding environment for errors, and `args` is a binding environment representing the actual arguments of the function call. To access argument number `i` use the C macro:

```
nthargval(args,i)
```

The arguments are enumerated from 1 and up.

The C function

```
int envarity(bindenv args)
```

returns the actual arity of the function call.

For example, the following *aLisp* function `sumfn()` adds an arbitrary number of integer arguments:

```
oidtype sumfn(bindtype args,bindtype env)
{
   int sum=0, arity = envarity(args), i, v;

   for(i=1;i<=arity;i++)
   {
      IntoInteger(nthargval(args,i),v,env);
      sum = sum + v;
   }
   return mkinteger(sum);
}
```

Variable arity functions are the registered to the system with `extfunctionn()`:

```
extfunctionn("SUM",sumfn);
```

The Lisp function LIST has the following implementation:

```
oidtype listfn(bindtype args,bindtype env)
{
  dcl_oid(res);
  int arity=envarity(args), i;

  for(i=arity;i>=1;i--)
  {
     a_setf(res,cons(nthargval(args,i),res));
  }
  a_return(res);
}
```

Notice how the iteration over the arguments is done in reverse order to get the correct list element order.

2.1.3. Defining special forms

Special forms are external Lisp functions whose arguments are not evaluated by the *aLisp* interpreter when the C implementation function is called.

C functions implementing special forms have the signature:

```
oidtype fn(bindtype args,bindtype env)
```

Analogous to variable arity functions the macros `envarity()` and `nthargval()` can be used to investigate the actual arguments. The difference is that `nthargval()` here returns the *unevaluated* value, unlike for variable arity functions where evaluated values are returned.

For example, the following C function implements the aLisp special form `QUOTE`:

```
oidtype quotefn(bindtype args, bindtype env)
{
       return nthargval(args,1);
}
```

Special forms are registered using `extfunctionq()`:

```
extfunctionq("QUOTE",quotefn);
```

For evaluating unevaluated forms this system function can be used:

```
oidtype evalfn(bindtype env, oidtype form)
```

For example, the following C function implements the special form `(WHILEA PRED FORM1 FORM2 ...)` that iteratively executes FORM1 etc. while PRED is non-nil:

```
oidtype whileafn(bindtype args, bindtype env)
{
   dcl_oid(cond), dcl_oid(v);
   int arity = envarity(args), i;

   a_setf(cond,nthargval(args,1));
   for(;;)
   {
      a_setf(v,evalfn(env,cond)); /* Evaluate condition */
      if(v == nil)  /* Condition false */
      {
         a_free(v); /* Release v and cond before returning */
         a_free(cond);
         return nil;
      }
      for(i=2;i<=arity;i++)
          {
             a_setf(v,evalfn(env,nthargval(args,i)));
          }
   }
}
```

Notice that `v` and `cond` must be released before the function is exited. Furthermore, the above definition is not fully correct, since if `evalfn()` fails because of some logical error in the evaluated form, an error will be thrown which will make `evalfn()` never return. Thus, in case of an error in the evaluation, the storage referenced by `v` and `cond` will never be deallocated. Another version of `whilea()` which also manages this memory deallocation correctly will be presented in the next section.

2.2. Error management in C

aLisp has its own error management system integrated with the storage manager. In order for the storage manager to correctly release data after failures, abnormal function exits should always use the system error management, rather than e.g. directly calling C or C++ error management.

2.2.1. Unwind Protection

To unconditionally catch failed operation the *unwind protect* mechanism is used. This is necessary sometimes to guarantee that certain actions are performed even if some called function terminates abnormally. For example, space may need to be deallocated or files be closed. For this purpose the system provides an *unwind-protect* feature in C, similar to what is provided in *aLisp* [2]. Unwind protection is provided through the following three macros:

```
{unwind_protect_begin;  /* Always new block */
     main code
   unwind_protect_catch;  /* This statement MUST ALWAYS be executed */
     unwind code
 unwind_protect_end;}   /* Will continue abnormal evaluation */
```

The `main code` is the code to be unwind protected. The `unwind code` is always executed both if the main code fails or succeeds. In the unwind code, a flag, `unwind_reset`, is set to `TRUE` if the code is executed as the result of an exception. The unwind code is executed outside the scope of the current unwind protection. Thus, exceptions occurring during the execution of unwind code is unwound by the next higher unwind protection.

WARNING: The `unwind_protect_end` code *must* be executed; never return directly out of the main code block. If `unwind_protect_end` is not executed after an exception, then the exception is not continued. Always execute `unwind_protect_end`, unless you want to catch all possible exceptions.

For example, a correct version of while that releases memory also in case of an error in the evaluation can be defined as follows:

```
oidtype whilebfn(bindtype args, bindtype env)
{
   dcl_oid(cond), dcl_oid(v);
   int arity = envarity(args), i;

   {unwind_protect_begin
      a_setf(cond,nthargval(args,1));
      for(;;)
      {
         a_setf(v,evalfn(env,cond)); /* Evaluate condition */
         if(v == nil)  /* Condition false => exit for loop */
            break;
         for(i=2;i<=arity;i++)
         {
            a_setf(v,evalfn(env,nthargval(args,i)));
         }
      }
    unwind_protect_catch;
      a_free(v); /* Release v and cond before exiting function */
      a_free(cond);
    unwind_protect_end;
    return nil; /* This statement not executed in case of an error */
```

```
    }
}
```

WARNING: Some compilers (e.g. *gcc*) may not restore local variables correctly when an exception has occurred unless they are defined as *volatile*. The macro `dcl_oid(x)` declares `x` as a volatile variable initialized to `nil`.

2.2.2. Raising errors.

Every kind of error has an *error number* and an associated *error message*. There are predefined error numbers for common errors defined in `storage.h`. To raise an *aLisp* error condition use the system function:

```
oidtype lerror(int no, oidtype form, bindtype env);
```

`no` is the error number.

`form` is the failed expression.

`env` is the binding environment for the error.

For example, the following code implements the Lisp function CAR:

```
oidtype carfn(bindtype env, oidtype x)
{
  if(x==nil) return nil; // (CAR NIL) = NIL
  if(a_datatype(x) != LISTTYPE) return lerror(ARG_NOT_LIST,x,env);
  return hd(x);
}
```

A few convenience macros for common error checks are defined in `storage.h`:

`OfType(x,tpe,env)`	Raise a standard error if x is not of type tpe.
`IntoString(x,into,env)`	Set the variable `into` (declared `char* into`) to a *copy* of the text of a symbol or string object `x` (declared `oidtype x`). The copy is pushed on the C stack and automatically freed when the C function is exited.
`IntoInteger(x,into,env)`	Convert numeric object x into C integer.
`IntoDouble(x,into,env)`	Convert numeric object x into C double.

To register a new error to the system use:

```
int a_register_error(char *msg);
```

`a_register_error` gets a unique error number for the error string msg. If `msg` has been registered before its previous error number is returned.

2.3. Calling Lisp from C

An *aLisp* function can be called from C by using the following C function:

```
oidtype call_lisp(oidtype lfn, bindtype env, int arity,
                  oidtype a1, oidtype a2,...)
```

`lfn`	is the *aLisp* function to call.
`env`	is the error binding environment.
`arity`	is the actual arity of the call.
`a1,a2,...`	are the actual arguments of the call.

For example, the following code implements an *aLisp* function (MYMAP L FN) which applies FN on each element in L:

```
oidtype mymapfn(bindtype env, oidtype l, oidtype fn)
{
   dcl_oid(res), dcl_oid(lst);

   {unwind_protect_begin;
     a_setf(lst,l);
     while(listp(lst))
     {
       a_setf(res,call_lisp(fn,env,1,hd(lst)));
       a_setf(lst,tl(lst));
     }
    unwind_protect_catch;
     a_free(res);
     a_free(lst);
    unwind_protect_end;
   }
   return nil;
}
```

Notice that unwind protection has to be used here to guarantee that the temporary memory locations are always released even if the call to `fn()` causes an *aLisp* error.

Also notice that the called *aLisp* function might allocate new data objects and these have to be freed correctly by assigning `res` using `a_setf()` and always releasing `res` when the function is exited.

The use of symbols is convenient for calling named *aLisp* functions from C. For example, the following function prints each element in a list:

```
oidtype mapprintfn(bindtype env, oidtype l)
{
   dcl_oid(printsymbol), dcl_oid(lst);

    printsymbol = mksymbol("print");
   a_setf(lst,l);
   while(listp(lst))
   {
      call_lisp(printsymbol,env,1,hd(lst));
      a_setf(lst,tl(lst));
   }
   return nil;
```

```
}
```

Notice that symbols like PRINT are permanent and when a symbol is referenced from a location it need not be reference counted as in the assignment of `printsymbol` above. Also the call to PRINT is guaranteed to not generate any new objects and need not be released.

To call Lisp functions with variable arity use:

```
oidtype apply_lisp(oidtype fn, bindtype env, int arity, oidtype args[]);
```

The difference to `call_lisp()` is that the arguments are passed in the array `args`. Don't forget to release the result.

To evaluate a C string of Lisp forms use:

```
oidtype eval_forms(bindtype env, char *forms);
```

All forms in `forms` are evaluated. The value of the last evaluation is returned as value. Don't forget to release the result.

2.3.1. Direct C calls

If the name of a C function implementing an *aLisp* function is known, it is more efficient to directly call the C function than to use `call_lisp()`. However, arguments and results of direct C calls must be handled carefully to avoid storage leaks. The automatic deallocation of temporary storage is NOT performed with direct C function calls. For example, the following correctly defined external Lisp function prints 'hello world' by directly calling the *aLisp* function PRINT:

```
oidtype hellofn(bindtype env)
{
   dcl_oid(msg);

   a_setf(msg, mkstring("hello world"));
   printfn(env, msg, nil); // PRINT has two arguments
   a_free(msg);
   return nil;
}
```

By contrast, the following incorrect implementation would cause a storage leak because the 'hello world' string is not deallocated:

```
oidtype hellofn(bindtype env)
{
   printfn(env, mkstring("Hello world"), nil);
   return nil;
}
```

Notice that `call_lisp()` automatically garbage collects its arguments upon return; thus temporary objects among the arguments are automatically freed. For example, the following

definition of `myhello()` would be correct but slower than the previous definitions:

```
oidtype hellofn(bindtype env)
{
   call_lisp(mksymbol("print"),2,env, mkstring("Hello world"), nil);
   return nil;
}
```

2.4. C functions for debugging

The reference counter of a physical storage object referenced by a handle is obtained with:

```
int refcnt(oidtype x)
```

Any *aLisp* object can be printed on the standard output with:

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

When defining new physical storage type it is important to make sure that object allocation and deallocation works OK. Therefore there is a facility in the Amos II and *aLisp* top loops to trace how many objects are allocated, or deallocated, respectively. Turn on that facility by evaluating the form

```
(STORAGESTAT T)
```

The system will then make a report of how many objects have been (de)allocated for each physical storage type. Make sure that the same number of objects is deallocated as allocated if that is expected. Notice that object references might be saved in the database log and therefore you should rollback database updates when necessary to get the balance between allocated and deallocated objects.

Turn off storage usage tracing with:

```
(STORAGESTAT NIL)
```

In C memory leaks can be traced also by calling the system function:

```
void a_printstat(void)
```

It prints a report on how much storage was allocated since the previous time it was called.

2.4.1. Trapping memory corruption

When adding C-code to the system it may happen that the database image accidently becomes corrupted, meaning that some handle references some illegal location. If not all the conventions for writing C-code are not systematically followed errors typically occur in a completely different place of the system. When the system finds a corrupted memory location in the image it

will print an error message:

```
Memory corruption in location 134000 (= 12345)
```

The two numbers *134000* and *12345* indicate that memory location denoted by handle (oidtype) *134000* is corrupt and points to a word containing the integer *12345*. To trap this when it actually happens can be done by calling the function

```
a_setdemon(oidtype loc, int val)
```

for example

```
a_set_demon(134000, 12345);
```

It causes the *aLisp* interpreter to continuously check if *loc* is equal to *val*. Whenever *loc* becomes equal to *val* an error is raised and the demon is turned off.

2.5. Interrupt handling

The interrupt handling system is managed by the *aLisp* function (CATCHINTERRUPT). This function is called whenever an interrupt has occurred. It either prints a message or catches the interrupt. The following C macro checks if an error has occurred and calls CATCHINTERRUPT if that is the case:

```
CheckInterrupt;
```

An interrupt is indicated when the global C variable `InterruptHasOccurred` is set to `TRUE`. `CheckInterrupt` is called by the *aLisp* interpreter after every function call. If you write long-running C code you should insert calls to `CheckInterrupt` to allow interrupts to be managed.

References

1 Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld: *Amos II Release 11 User's Manual*, http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html

2 Tore Risch: *aLisp v2 User's Guide*, UDBL, Dept. of Information Technology, Uppsala University, http://user.it.uu.se/~torer/publ/alisp2.pdf, 2006

3 T.Risch, V.Josifovski, and T.Katchaounov: *Functional Data Integration in a Distributed Mediator System* in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003, http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf.

Index