sa.Storage 2.0

A main-memory storage manager

Stream Analyze Sweden, AB

2022-02-25

The *sa.engine* system uses a main memory database storage manager called *sa.Storage*. Both data and models in an sa.engine system is stored in a *database image* managed by sa.Storage. The storage manager is scalable allowing data structures to dynamically and gracefully grow very large without performance degradation or lags. The system includes a real-time 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, called *storage types*, managed by the system. An extensible byte stream mechanism allows news kinds of I/O and communication infrastructures to be plugged in without kernel code changes. sa.Storage is tightly integrated with a Lisp system called *aLisp*. New aLisp data types can be defined in C and made interoperable between Lisp and C. This report documents sa.Storage. It also explains how to extend aLisp with new datatypes and functions.

Table of contents

1. Introduction

sa.Storage is a main memory storage manager that represents both data and models in sa.engine. A central component of sa.engine is a main memory database managed by sa.Storage. Figure 1 illustrates the components of sa.Storage.

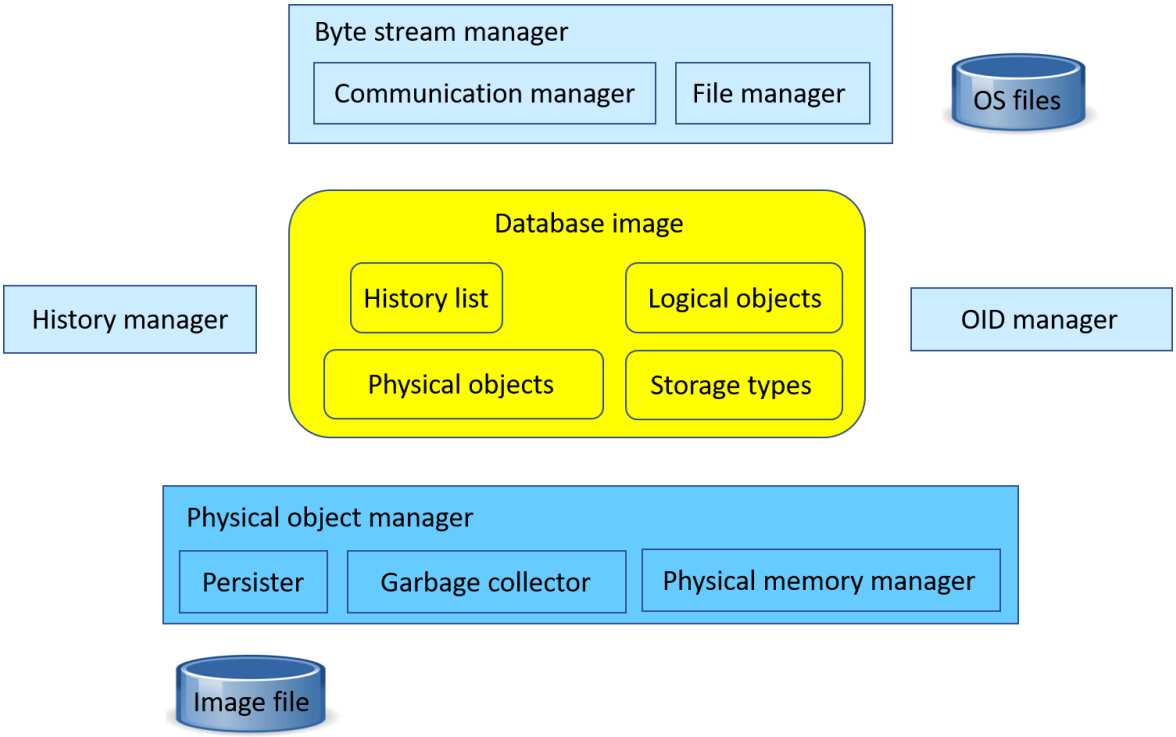

Figure 1: Components of the storage manager

With sa.Storage the C programmer has the choice of allocating physical objects persistently by using a set of primitives provided by the storage manager. Persistency in this case means that physical objects are allocated inside a memory area called the *database image*, which is a main memory area where the database resides. C structures allocated inside the database image are called *physical objects*.

The *physical object manager* manages the physical objects. In particular the *physical memory manager* is responsible for allocating and freeing memory areas inside the database image. A real time *garbage collector* instructs the physical memory manager to free physical objects no longer used. The database image can be saved on disk in an *image file* using the *persister* by the C function `a_rollout(char *filename)` or the Lisp function `ROLLOUT`. The image is restored when restarting the system with the image file as command line argument or when initializing the system from an embedder passing the name of the image file as parameter in `sa_engine_init(int argc, char **argv)` [3].

The physical object manager is extensible allowing the programmer to register new kinds of

physical objects called *storage types* as plug-ins without kernel code changes. All physical objects are tagged with a numeric storage type identifier. There are a number of built-in basic storage types and new ones can easily be added in C.

When physical objects are changed the *history manager* can optionally be used to record state changes of the updated object before and after the update in a *history list* inside the database image. The history list allows to restore the old state before the update in case of errors. Since the history list is implemented as a main memory linked list there is a very low performance penalty in using the history list. Nevertheless, it will occupy some space and the programmer therefore has the option to not use the history list when state changes need not be recorded.

sa.Storage can be extended in several ways by hooking up C functions without kernel changes:

1. Custom *storage types* can be implemented by registering a number of C functions for each new storage type to implement allocation, dealloction, printing, garbage collection, etc.

2. Custom *byte streams* can be implemented as plugins. A byte stream is a conventional stream of bytes as used in, e.g., C-based file systems. This makes the system agnostic to underlying file and communication infrastructures. By default byte streams are defined for C-files, TCP sockets, TLS sockets, and main memory C strings. It is possible to add new byte streams as C plugins for custom file and communication systems without changing any kernel code.

3. For each storage type custom *linearization* and *delinearization* functions can be plugged in without any code changes. Once the (de-)linearization functions are defined the physical objects can be written into file or communication streams in such a way that they are recreated when read back or received. There can be different (de-)linearization for different byte streams; for example, arrays are printed binarily to files and sockets but textually (human readable) on standard output.

The storage manager is a separate subsystem, which is independent of the rest of the sa.engine system. There are several sa.engine system layers on top of the storage manager. An important layer is a Lisp interpreter, *aLisp*,which is a Lisp interpreter tightly interfaced with sa.Storage. A large part of sa.engine is written in aLisp. The use of the aLisp system is documented separately in [1], while this document includes a description of how to extend aLisp with new data types and functions written in C. The datatypes of aLisp are all implemented as storage types. The sa.Storage system itself is independent of aLisp.

Another important service of the storage manager is to provide a *garbage collector* that automatically deallocates memory in the database image that is no longer used.

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. Unlike storage objects, the programmer is responsible for deallocating transient data manually since C has no automatic garbage collector. A particular problem is thereby handling references between persistent and transient data since

transient data is lost when the system is restarted. For this, there is a special mechanism to restore transient data when the system is restarted.

While physical data objects are C record structures stored in the database image, *logical objects* represent object used in OSQL. Only logical objects can be used in OSQL, while physical objects can be manipulated in C/C++ or aLisp. The logical data objects are internally represented by one or several physical objects. For example, OSQL objects of logical data type INTEGER are directly represented by a physical object having a storage type 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 objects of type FUNCTION, are represented by data structures consisting of several physical objects of different types. Logical objects in OSQL having explicit object identifiers, called *surrogate objects*, are represented by the storage type named `OID` with type tag `SURROGATETYPE`. Physical objects having storage type named `OID` describe the properties of logical surrogate object. One property of an OID object is a numeric identifier (the OID number) maintained by the *OID manager*; another one is the OSQL type of the logical object.

1.1. Object handles

All accesses to physical objects is made through *object handles,* which are indirect identifiers for physical data records in C inside the database image. The representation of object handles is currently unsigned 32-bit integers, but the system can be reconfigured for different kinds of object handle representations (e.g. 64-bits integers or pointers). 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 `sa_storage.h`.

Object handles are declared as C type `ohandle` and initialized to the global C constant `nil`.

1.2. Physical Objects

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

```
struct integercell
{
  objtags tags;
```

```
  HEADFILLER;
  char data[8];
};
```

The field `tags` is used by the system, the field `data` stores the actual 64-bits integer value, and `filler` aligns the value to a full-word.

The header of a physical object (field `tags` with C 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 storage type has an associated *storage type identifier* number and a unique *storage type name* string known to the storage manager. The main memory array `typefns` represents information about the storage types. Since the type identifier is represented by one byte there can be up to 256 physical types defined. A number of (currently 20) physical storage types are predefined, including LIST, SYMBOL, INTEGER, REAL, EXTFN (foreign aLisp functions), CLOSURE (aLisp closures), STRING, ARRAY (1D fixed size arrays), STREAM (file streams), TEXTSTREAM (streams to text buffers), HASHTAB (hash tables), and BINARY (bit strings). In `sa_torage.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 storage 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 storage types are defined as C macros in `sa_torage.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 storage types through the C function `a_definetype`, explained below. It defines to the storage manager the properties of the new storage type.

1.3. Dereferencing

In order to access or change the contents of the physical object for a handle, the handle has to be converted into a C pointer to the physical object in the database image. This process is called to *dereference* the handle. The dereferencing of object handles 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 data in the image may move when new data is allocated, so the programmer can cache direct pointers to physical objects through dereferencing **only** 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 unless you know

that the dereferenced object cannot move between accesses.

The following C macro dereferences a handle:

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

`dr` returns the address inside the database image of the physical object 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 value of the integer is accessed with `dr(ic,integercell)->data`. The structure `integercell` represents 8-byte (64-bits) integers. The macro `getinteger(h)` dereferences `h` as a C 64-bits integer (type named `LONGINT`) while `getinteger32(h)` casts the integer to C-type `int`.

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

```
void printint(ohandle h)
{
   struct integercell *dh = getinteger32(h);
   printf("The integer is %d\n", dh);
}
```

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

```
a_datatype(h)
```

returns the type identifier of a handle `h`.

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

```
void printint2(ohandle h)
{
   if(a_datatype(h) == INTEGERTYPE)
      printf("The integer is %d\n",getinteger32(h));
   else printf("It is not an integer\n");
}
```

WARNING: Storage manager operations may invalidate C pointers to physical objects in the database image since the physical objects might move to other memory locations when the image is expanded. Thus, dereferenced C-pointers may become incorrect once a system call is made that causes the image to expand. Physical 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. Also, if the current C-thread is unlocked, some other thread my invalidate dereferenced handles.

1.4. Assigning handles to locations

In order for the storage manager and garbage collector to function correctly, a C location `h` (variable or field) of type `ohandle` *must* be initialized to the global variable `nil` by declaring it:

```
ohandle h=nil;
```

To update the location the following C macro *must* be used:

```
a_setf(loc,h);
```

`a_setf(loc,h)` corresponds to an assignment of the C location `loc` (i.e. `loc` is a C variable or field of type `ohandle`) to the object handle `h`, i.e. `loc=h`, but, unlike an assignment, `a_setf` handles reassignments of locations correctly, `a_setf(loc, h)` decreases the reference count of the handle *previously* referenced from `loc` and increases the reference counter of `h` The reference counter increment of `h` indicates to the system that there is some location (i.e. `loc)` that now holds a reference to the physical object `h` and it therefore cannot be deallocated until the location is *released*, meaning that the location `loc` does not need to access the object any more. A location `loc` is released with the C macro:

```
a_free(loc)
```

After calling `a_free(loc)` the handle in `loc` will not be physically removed from the database image if there is some other location still holding a reference to it. No other location holds a reference to a physical object if the reference counter is 0. Thus, when the reference counter is decreased to 0 by `a_free()` or `a_setf()` the physical object is passed to the garbage collector for deallocation from the image. Thus, unlike the C function *free(loc)*, `a_free(loc)` will deallocate `x` *only* when there is no other location holding a reference the object handle in `loc`.

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

Notice that the location *must* be previously 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 to `nil` by declaring

```
ohandle loc=nil;
```

before calling `a_setf(loc,..)`. An alternative is to use the macro `a_let(loc,h)` the *first* time a location is assigned a handle. It is similar to `a_let()` but assumes that the old value of `loc` was uninitialized and will therefore only increase the reference counter of `h`, while ignoring the old value in `loc`. This code

```
ohandle loc;
a_let(loc,h);
```

is equivalent to:

```
ohandle loc=nil;
a_setf(loc,h);
```

1.5. 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.

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

Type-name	Type tag	Short description
`LIST`	`LISTTYPE`	Linked lists
`SYMBOL`	`SYMBOLTYPE`	Symbols
`INTEGER`	`INTEGERTYPE`	64-bits integers
`REAL`	`REALTYPE`	64-bits floating point numbers
`STRING`	`STRINGTYPE`	Strings
`ARRAY`	`ARRAYTYPE`	1-D arrays (vectors) of handles
`OID`	`SURROGATETYPE`	Surrogate objects
`STREAM`	`STREAMTYPE`	File streams of bytes
`TEXTSTREAM`	`TEXTSTREAMTYPE`	Byte streams over strings
`SOCKET`	`sockettype`	Byte stream over sockets
`HASHTAB`	`HASHTYPE`	Hash tables
`HASHBUCKET`	`HASHBUCKETTYPE`	Internal buckets of hash tables
`LOGRECORD`	`logrecordtype`	Update events in history lists
`EXTFN`	`EXTFNTYPE`	Foreign Lisp function in C
`CLOSURE`	`CLOSURECELL`	Lisp closure

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:

```
{
   ohandle mystring=nil;
   ...
   a_setf(mystring,mkstring("Hello world"))
   ...
   a_free(mystring);
};
```

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

```
{
   hhandle mystring=nil;
   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 storage types:

`ohandle mkinteger(int i)`	(macro) Construct handle for a new integer
`int integerp(ohandle h)`	(macro) `TRUE` if `h` is a handle for an integer
`int getinteger(ohandle h)`	(macro) Dereference a handle for an integer
`ohandle mkreal(double r)`	(macro) Construct handle for a new real
`int realp(ohandle h)`	(macro) `TRUE` if `h` is a handle for a real
`double getreal(ohandle h)`	Dereference a handle for a real
`ohandle mkstring(char *s)`	(macro) Create handle for a new string
`int stringp(ohandle s)`	(macro) `TRUE` if `h` is a handle for a string
`char *getstring(ohandle s)`	(macro) Dereference a handle for a string
`ohandle new_array(int size,ohandle init)`	Construct handle for a new array with elements `init`
`int arrayp(ohandle h)`	`TRUE` if `h` is a handle for an array
`int a_arraysize(ohandle arr)`	return the array size
`ohandle a_seta(ohandle arr,int pos,ohandle val)`	Set an array element
`ohandle a_elt(ohandle arr,int pos)`	Retrieve array element
`ohandle a_vector(ohandle x1,...,xn,NULL)`	Create a new array and its elements `x1 ... xn`.
`ohandle cons(ohandle x,ohandle y)`	Create handle for a new list cell
`int listp(ohandle h)`	(macro) `TRUE` if `h` is a list cell
`ohandle hd(ohandle h)`	(macro) Head of list cell
`ohandle tl(ohandle h)`	(macro) Tail of list cell
`ohandle a_list(ohandle x1,...,xn,NULL)`	Create new list of `x1 ... xn`
`ohandle mksymbol(char *x)`	(macro) Create a new symbol
`int symbolp(ohandle h)`	(macro) `TRUE` if `h` is a symbol
`ohandle globval(ohandle h)`	(macro) Get global value of symbol.
`char *getpname(ohandle h)`	(macro) Get print name of symbol
`a_print(ohandle x)`	Print object of any type. Very useful for debugging.
`ohandle t`	Symbol `T` representing TRUE
`ohandle nil`	Symbol `NIL` representing empty list and FALSE

For example, the following C function adds two integers:

```
ohandle add(ohandle x, ohandle y)
{
    LONGINT sum;
```

```
   if(a_datatype(x) != INTEGERTYPE ||
      a_datatype(y) != INTEGERTYPE) {
      printf("Cannot add non-integers\n");
      exit(1);                    ← Should call error manager here.
   }
   sum = getinteger(x) + getinteger(y);
   return mkinteger(sum);
}
```

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

```
ohandle x=nil, y=nil, s=nil; // 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 as sum of a x and y
printf("The sum is %d\n",getinteger(s));
a_free(s);             // release locations s, x, y
a_free(x);
a_free(y);
```

Notice that the datatype of an object handle should always be checked before it can be dereferenced. It will later be shown how to utilize the exception system of sa.engine when type errors occur.

1.6. Defining storage types

This subsection describes how to introduce new storage types to sa.Storage. This is required when new C data are defined for aLisp.

The include file `sa_storage.h` contains record templates for each storage type.

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

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

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

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

`print_function(ohandle h, ohandle str, int mode)` is a *print function* to provide a customized printing into the stream `str` of physical objects of the new type. A default print function is called if it is `NULL`. See section 1.7.1.

1.7. Byte streams

sa.Storage has several data types representing byte streams:

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

The following system standard streams are defined:

`ohandle stdinstream` C's standard input stream
`ohandle stdoutstream` C's standard output stream

Streams are represented as physical objects with special *stream attributes* in the structure `streamheader` stored in physical objects after the tags in beginning of the template. For example, the storage type `STREAM` for file steams has the template:

```
struct streamcell   /* OS file streams */
{
  objtags tags;
  struct streamheader header;
  int opened;       /* TRUE while file opened */
  int tailed;       /* TRUE if file is tailed */
  FILE *fp;         /* OS file pointer */
};
```

The structure `streamheader` has the template:

```
struct streamheader
{
  short int bytes;  /* Total size of object in bytes, incl. header */
  char autoflush;   /* Flush after each item and new line */
  char systime;     /* Maintain current systime */
  char newline;     /* True when \n is just printed */
  char filler;
  int line_num;     /* Current line number */
  ohandle logstream;/* Stream to copy input to if non-NIL */
  ohandle origin;   /* ID of sender of data if known or nil */
  ohandle destination;/* ID of receiver of data if known or nil */
};
```

The `header` field must always be present for stream templates. Additional specific attributes can be added after the end of the stream header. Once a storage type has been defined using `definetype()` it 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)(ohandle),
                                   int(*ungetc)(int,ohandle),
                                   int(*feof)(ohandle),
                                   int(*puts)(char*,ohandle),
                                   int(*putc)(int,ohandle),
                                   int(*fflush)(ohandle),
                                   int(*fclose)(ohandle));
```

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

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

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

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

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

`Int readbytes(ohandle 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(ohandle 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(ohandle stream)` Flush stream buffer contents.

`int fclose(ohandle 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(ohandle stream);`	Read one character
`int a_ungetc(int c, ohandle stream);`	Unread one character
`int a_puts(char *str,ohandle stream);`	Write string
`int a_writebytes(ohandle stream, void *buff, unsigned int len);`	Write block
`int a_putc(int c, ohandle stream);`	Write a character
`int a_puti(LONGINT i, ohandle stream);`	Write an integer
`int a_putr(double i, ohandle stream);`	Write a real number
`int a_readbytes(ohandle stream, void *buff, unsigned int len);`	Read block
`int a_fclose(ohandle stream);`	Close stream
`int a_feof(ohandle stream);`	Test for end-of-file
`int a_fflush(ohandle stream);`	Flush stream buffer

The performance of stream management can be substantially 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 since it will be done byte-by-byte.

1.7.1. Marshalling objects

Streams are usually 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 sa.engine peers, e.g. using sockets. The function `a_printobj(handle h, handle str)` writes the physical object `h` on a stream `str` in such a format (S-expression) that a copy of the object is allocated when the function `a_read(handle str)` reads the object from the same stream. Thus *a_printobj()* and *a_read()* are sa.Storage's generic (de-)marshalling functions. They use Lisp's S-expressions to provide standardized marshalling and demarshalling for the built-in storage types. Customized (de-)marshalling should 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:

`ohandle a_read(ohandle stream)`
: Read (unmarshal) S-expression from a stream. This corresponds to the Lisp function READ.

`ohandle a_print(ohandle s)`
: Print S-expression a followed by a line feed on `stdoutstream`, normally for debugging.

`ohandle a_printobj(ohandle s, ohandle stream)`
: Print S-expression s followed by a line feed as delimiter on stream. This corresponds to the Lisp function PRINT.

`ohandle a_prin1(ohandle s, ohandle stream, int princflg)`
: Print S-expression s on stream. If `princflg` is FALSE the printout be marshalled using the escape character `\` when necessary to allow for subsequent reading; if `princflg` is TRUE object will be written without escapes 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.

`ohandle a_terpri(ohandle 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 and sa.engine can be utilized in C for uniform and efficient error management.

In order to interface aLisp with C/C++ you must include the file `sa_lisp.h` in your C program. 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 standard output. It has the C implementation:

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

The include file `sa_lisp.h` contains all necessary declarations for implementing external Lisp functions in C. External Lisp function definitions must always return handles of type `ohandle`. 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 can be done in a main C program, the *driver program*, after the system has been initialized after `sa_engine_init(argc, argv)` is called, or after a DLL or shared object library is loaded dynamically. The following driver program initializes the system, registers HELLO, and calls the aLisp read-eval-print loop (REPL) with prompter string 'Lisp>'.

```
#include "sa_lisp.h"

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

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

When the above program is run the user can call HELLO from the REPL by typing

```
(hello)
```

Foreign Lisp functions can also be defined when loading C plugins (DLLs or shared objects) to sa.engine or *sa.core* se documention of plug-ins in C.

2.1.1. Defining foreign Lisp functions in C

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

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

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

For example, the following function implements a foreign Lisp function to add two numbers:

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

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

`addfn` is registered with

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

The number ’2’ after ’`extfunction`’ indicates that this foreign function takes two arguments.

Foreign Lisp functions should always check the legality of the handles they receive, otherwise the system may crash. To check that a handle `h` is of an expected storage type (i.e. Lisp type) use the C macro:

```
OfType(h,tpe,env)
```

A standard error will be generated if `h` does not have the storage type tag `tpe`. For integers the above used macro `IntoInteger(h, i, env)` is a convenient alternative to `OfType`. It safely dereferences handle `h` to integer `i`.

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

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

Where `name` is the Lisp name of the foreign function and `cfn` is a pointer to its implementation in C.

Different versions of `extfunctionX()` are available depending on the arity X of the external

Lisp function. For example,

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

There are corresponding registration functions for foreign 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(h);
```

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

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

```
ohandle add3fn(bindtype env, ohandle x, ohandle y, ohandle z)
{
   ohandle s=nil;

   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 `s` instead had been returned by the C statement

```
   return s;
```

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

The following function reverses a list:

```
ohandle myreversefn(bindtype env, ohandle l)
{
   ohandle lst=nil, res=nil;

   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 `myreverse` with:

```
extfunction1("myreverse", myreversefn);
```

WARNING: You *cannot* assign C function parameters (such as `l` in the example) with `a_setf(l,..)` or release them with `a_free(l)`, since 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()`.

WARNING: The C implementation of a foreign Lisp function must always return a legal handle, otherwise the system might crash. It is therefore recommended to run the system in 'debug mode' (by calling `(debugging t)`) while testing external Lisp function so that the system checks the legality of data passed between Lisp and C.

2.1.2. Variadic foreign Lisp functions

Variadic external functions accept any number of arguments. Foreign Lisp functions with more than six arguments need to be defined as variadic functions. Variadic foreign Lisp functions have the signature:

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

where `env` is the binding environment for raising 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 Lisp function `sumfn()` adds an arbitrary number of integer arguments:

```
ohandle sumfn(bindtype args,bindtype env)
{
   LONGINT sum=0;
   int 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:

```
ohandle listfn(bindtype args,bindtype env)
{
  ohandle res=nil;
  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:

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

Analogous to variadic foreign 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 variadic functions where evaluated values are returned.

For example, the following C function implements the Lisp special form `quote`:

```
ohandle myquotefn(bindtype args, bindtype env)
{
      return nthargval(args,1);
}
```

Special forms are registered using `extfunctionq()`:

```
extfunctionq("myquote",myquotefn);
```

For evaluating unevaluated forms this system function can be used:

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

For example, the following C function implements the special form `(mywhile pred form1 form2 ...)` that iteratively executes `form1` etc. while `pred` is non-nil:

```
ohandle mywhilefn(bindtype args, bindtype env)
{
   ohandle cond=nil, v=nil;
   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()` abort. Thus, in case of an error in the evaluation, the storage referenced by `v` and `cond` will never be deallocated. Another version of `mywhile()` which also manages this memory deallocation correctly will be presented in the next section.

2.2. Error management in C

sa.engine 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 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 often necessary 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 Lisp. Unwind protection is provided through the following three macros:

```
unwind_protect_begin;  /* New unwind-protected block */
```
main code

```
unwind_protect_catch;  /* This statement MUST ALWAYS be executed */
    unwind code
unwind_protect_end;   /* Will handle thrown exceptions */
```

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 the unwind code is thrown to the next higher unwind protection.

Notice that he `unwind_protect_catch` code *must* be executed; never return directly out of the main code block.

Notice that omitting `unwind_protect_end` will cause a compiler warning, so that if you want to catch all exceptions use `unwind_protect_cancel` instead of `unwind_protect_end`.

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

```
ohandle mywhilefn(bindtype args, bindtype env)
{
   volatile ohandle cond=nil, v=nil;
   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 is not executed in case of an error
}
```

Notice that some compilers (e.g. *gcc*) may not restore local variables correctly when an exception has occurred unless they are defined as *volatile*.

2.2.2. Throwing errors.

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

```
ohandle a_throw_errorno(bindtype env, int no, ohandle form);
```

`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`:

```
ohandle mycarfn(bindtype env, ohandle x)
{
  if(x==nil) return nil; // (car nil) = nil
  if(a_datatype(x) != LISTTYPE)
     return a_throw_errorno(env,ARG_NOT_LIST, x);
  return hd(x);
}
```

Alternatively error messages rather than error number can be thrown by calling the function:

```
ohandle a_throw_errormsg(bindtype env, const char *msg, ohandle form);
```

The following is equivalent to the above call to `a_throw_errorno()`:

```
 a_throw_errormsg(env, "Not a list", x);
```

Error messages are truncated to max 100 bytes.

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

`OfType(h,tpe,env)`	Raise a standard error if `h` is not of type `tpe`.
`IntoString(h,into,env)`	Set the variable `into` (declared `char* into`) to a *copy* of the text of a symbol or string object `h`. The copy is pushed on the C stack and automatically freed when the C function is exited.
`IntoInteger(h,into,env)`	Convert numeric object `h` into C `LONGINT` integer.
`IntoInteger32(h,into,env)`	Convert numeric object `h` into C `int`.
`IntoDouble(h,into,env)`	Convert numeric object `h` into C `double`.

To register a new error to the system use:

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

`a_register_error(msg)` gets a unique *error number* `no` for the error string `msg` to be used in `a_throw_errorno(env,no,x)`. If `msg` has been registered before its previous error number is returned. Some error numbers (such as `ARG_NOT_LIST`) are defined as macros in

sa_storage.h. The system handles dynamic error messages passed to sa_throw_errormsg() by assigning them the error number -1.

2.3. Calling Lisp from C

Lisp functions can be called from C by using the following C function:

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

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

For example, the following code implements a Lisp function (mymap l fn) that applies Lisp function fn on each element in list l:

```
ohandle mymapfn(bindtype env, ohandle l, ohandle fn)
{
   ohandle res=nil, lst=nil;

   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 the called Lisp 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.

Notice also 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 error exception.

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

```
ohandle mapprintfn(bindtype env, ohandle l)
{
   ohandle printsymbol=nil, lst=nil;
```

```
   printsymbol = mksymbol("print");
   unwind_protect_begin;
   a_setf(lst,l);
   while(listp(lst))
   {
      call_lisp(printsymbol,env,1,hd(lst));
      a_setf(lst,tl(lst));
   }
   unwind_protect_catch;
   a_free(lst);  // in case printsymbol fails
   unwind_protect_end;
   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:

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

The difference to `call_lisp()` is that the arguments are passed in the array `args`.

To evaluate a C string of Lisp forms use:

```
ohandle 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 Lisp function is known, it is more efficient to directly call the C function than to use `call_lisp()`. However, arguments and results of such direct C calls must be handled carefully to avoid storage leaks, since 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 Lisp function `print`:

```
ohandle hellofn(bindtype env)
{
   ohandle msg=nil;

   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:

```
ohandle 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 implementation:

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

2.4. C functions for debugging

There are a number of function useful for debugging C programs calling the sa.engine kernel. The most useful one is `a_print` that prints the S-expression representation of the objects referenced by handle `h`:

```
ohandle a_print(ohandle h)
```

You can also print using the OSQL format of `h` with:

```
int sa_print(ohandle h)
```

It returns an error code if `h` cannot be printed.

2.4.1. Tracing storage leaks

When defining a new storage type it is important to make sure that object allocation and deallocation work OK. Therefore, there is a facility in the OSQL and aLisp REPLs to trace how many objects are allocated, or deallocated, respectively. Turn on that facility by evaluating the Lisp form

```
(allocstat t)
```

The system will then make a report of how many objects have been (de)allocated for each storage type. Make sure that the same number of objects is deallocated and allocated if that is expected.

Notice that logged objects are not deallocated until `commit` or `rollback` is called. Therefore, you should turn off logging before tracing storage leaks by the command:

```
logging off;
```

Notice that the first time a Lisp call is made there may be just-in-time macro expansions and caches

that make the storage count balancing not match. Thus you should make one or two extra calls to "warm up" the system before tracing storage leaks.

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 the Lisp call:

```
(allocstat nil)
```

The C the function

```
a_allocstat(int clear)
```

does the same as `allocstat`. If the flag `clear` is `TRUE` the statistics is cleared without printing a report.

Notice that you normally have to "warm up" the system before using `a_allocstat()`.

If you have a leak that caused the image to grow continuously, you can trace what functions were called by calling the Lisp function:

```
(trace-expand flag depth)
```

If `flag` is true, it makes a Lisp function backtrace every time the high watermark of the database image is expanded. The depth of the backtrace is specified with `depth`. In this way you may get to know where the objects for a storage leak are allocated.

You can trace what Lisp functions were called when creating an object that was NOT deallocated during an evaluation by first calling:

```
(storage-trace flag depth)
```

Then do your evaluation and print backtraces of the Lisp functions allocating objects remaining after the evaluation by calling:

```
(print-storage-trace &optional file)
```

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

```
int refcnt(ohandle h)
```

When the reference counter of the object referenced by `h` is changed to 0 the garbage collector will call the finalizer of the object and mark it as deallocated by setting the reference counter to `DEALLOCREF`.

The following trapper calls the C function `trapper` when the object referenced by handle `h` is deallocated:

```
void a_trap_dealloc(ohandle h, void(*trapper)(ohandle))
```

2.4.2. Checking validity of handles

A common bug is that some handle is not properly initialized or that its memory has been overwritten. You can check the validity of a handle `h` by calling:

```
int illegal_handle(ohandle h, int circular_depth)
```

If the second argument is positive, `illegal_handle()` also tests whether `h` references a circular list structure down to the specified depth from `h`.

If the function returns a non-zero value it is an error code `no` that indicates how `h` is corrupted. The corresponding error message can be retrieved by calling:

```
char *a_errormessage(int no)
```

2.4.3. 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 all the conventions for writing C-code are not systematically followed errors typically occur in a completely different place of the system. For that reason one would like to know where in the C-code the memory is destroyed.

The C-macro

```
CI;
```

checks if the image is corrupted. If that is the case the system will print on what C source code line the corruption is detected along with a small explanation. Add calls to `CI` in the C-code where you suspect memory corruption occurs.

You can make the sa.engine interpreters continuously call `CI` by calling:

```
a_system_trust (0)
```

When the argument of `a_system_trust` is 0 the sa.engine kernel will continuously call `CI`. Calling `a_system_trust (0)` will signal memory corruptions in general with a significant performance overhead. The function has the signature:

```
int a_system_trust(int level)
```

It returns the *old* trust level. To just get the current trust level, call `a_system_trust(-1)`. The corresponding Lisp function is:

```
(system-trust level)
```

When calling `system-trust` from lisp the level is also saved in the image making it persistent, while the trust level of `a_system_trust` is not saved when the image is saved.

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 (ohandle) *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(ohandle 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 Lisp function `(catchinterrupt)`. This function is called whenever an interrupt has occurred. It either prints a message or catches the interrupt. The C macro `CheckInterrupt` checks if an error has occurred and calls `catchinterrupt` if that is the case.

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

If interrupt signal `signo` is raised in your C-program under Unix it will be caught if you beforehand have called:

```
(sig-bt signo)
```

Then the system will automatically print a backtrace of the C call stack and then abend. By default `(sig-bt 11)` is turned on under Unix to trap memory corruption.

For interactive trapping of signals you can call the function:

```
(sig-trap signo)
```

It will enter a special Lisp REPL when signal `signo` is raised. This REPL is not a regular Lisp break loop as in Section 7.1 of [1] since there are no break commands. Instead you can explicitly make a C backtrace by calling `(c-backtrace)` and a corresponding Lisp backtrace by calling `(backtrace)`. The current thread identifier is obtained with `(thread-id)`. To exit the REPL call the function `(exit)`. Call `(quit-now rc)` to abend sa.engine with return code `rc`.

References

[1] aLisp User's Guide, Version 2.0, Stream Analyze Sweden AB, 2020.

[2] Guy L. Steele Jr.: Common LISP the language, Digital Press, http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

[3] sa.engine *C Interfaces*, Version 2.0, Stream Analyze Sweden AB, 2020

[4] sa.engine *Java Interfaces*, Version 2.2, Stream Analyze Sweden AB, 2020,

[5] sa.engine *Lisp Interfaces*, Version 2.1, Stream Analyze Sweden AB, 2020.