sa.engine Lisp Interfaces

Tore Risch
Stream Analyze Sweden AB
Sweden

Version 2.1
2020-10-16
`sa_LispAPI_2.1.pdf`

A central component of the sa.engine system is a fully integrated Lisp interpreter called aLisp. It implements an extended subset of the programming language CommonLisp and is documented separately. This document describes the external interfaces between sa.engine and the embedded aLisp system. There are mainly two ways to interface sa.engine with Lisp programs: In the *client* interface Lisp programs call sa.engine, while in the *plugin* interface foreign OSQL functions are implemented as Lisp functions. The combination is also possible where foreign functions in Lisp and other languages call sa.engine back through the client interface.

Table of contents

1. Introduction

There are two main kinds of external interfaces, the *client* and the *plugin* interfaces:

- With the *client interface* Lisp functions call OSQL functions and queries in the host sa.engine of the embedded aLisp system. Furthermore, analogous *remote* OSQL queries and function calls can be shipped from aLisp to separate sa.engine servers.

- With the *plugin interface* OSQL functions are implemented as Lisp functions. The foreign Lisp functions are executed in the same process as the host sa.engine. The callin interface can be used also in foreign Lisp function implementations.

The result of an OSQL query or function call is an *object stream,* which is a possibly infinite stream of objects. The callin interface provides primitives to apply user *callback* aLisp functions on the elements in object streams. Analogously the callout interface provides API primitives to iteratively produce object streams.

This documentation introduces the aLisp sa.engine API through a number of example programs. You are assumed to be familiar with aLisp [1]. You can run the examples by starting sa.engine and typing the command *lisp;*, which will put you in an aLisp REPL. To go back to the OSQL REPL, enter the form :*osql*. For example:

```
sa.engine
[sa.engine] 1> lisp;
Lisp 1> (print ‘hello)
HELLO
Lisp 1> :osql
[sa.engine] 1> quit;
```

You can run aLisp forms from the command line by using the -*l* option, for example:

```
sa.engine -l ”(print ‘hello)(quit)”
```

You can run entire aLisp scripts in a file by the command line option -*L*, for example:

```
sa.engine -L myscript.file
```

2. The client interface

With the *client interface* there are two ways to call sa.engine from aLisp:

- In the *query interface* strings containing OSQL statements are dynamically compiled and evaluated. The result from a query is an object stream on which an aLisp callback function can be applied. The embedded query interface is relatively slow since the OSQL statements must be compiled and optimized at run time.

- In the *function interface* sa.engine functions are directly called from aLisp, without the overhead of dynamically compiling and optimizing OSQL statements. The function interface is significantly faster than the query interface. It is therefore recommended to always define sa.engine functions stored in the local database for the various sa.engine operations performed by the application and then use the function interface to invoke them directly.

2.1. Connections

When calling sa.engine from Lisp programs through the *client interface*, the application must be connected to some sa.engine peer. One such peer is the *embedded* sa.engine system that runs inside aLisp. This is called the *local connection*. An object representing the connection to the embedded sa.engine is obtained by calling:

```
(connection)
```

The Lisp program may also run as a client connected to a remote sa.engine *stream server* (SAS) running on the same or some other computer. This is called the *remote connection*. With the remote connection several applications running in different locations can remotely access the same SAS concurrently. The applications and the SAS run as separate programs so that the server will survive client crashes and vice versa. A connection object to a remote peer `p` is obtained by calling:

```
(connection p)
```

For example this for binds global variable `_mypeer_` to a connection to the peer named “mypeer”:

```
(defglobal _mypeer_ (connection "mypeer"))
```

A connection object `c` returned from `connection` holds the necessary information for calling sa.engine primitives in the remote peer or the embedded sa.engine system from Lisp. It is passed as an argument to other client interface functions, as explained later.

The peer `p` identifies which sa.engine system to connect to. It is specified as a string. A peer can be one of

a) the embedded sa.engine system,
b) an sa.engine *stream server*, *SAS*, coordinating communication with other peers,
c) an sa.engine *edge client* running on an edge device registered in a SAS, or
d) a *nameserver*, which is a SAS that keeps track of all peers in a federation of sa.engine peers.

The format of a peer identifier is one of:

```
""
"peer"
"peer@host"
"peer@host:portno"
```

The empty string "" creates a connection to the embedded sa.engine.
If just `"peer"` is specified the peer is the name of a local edge client or SAS known by the nameserver running on the same computer as the client. The local nameserver can be reached using the peer name `"nameserver"`.

If "`peer@host`" is specified a connection is established to a peer managed by the name server of the specified `host`. Specifying `peer@localhost` is equivalent to just `peer`.

The nameserver by default listens on port 35021; the format "`peer@host:portno`" is used when the nameserver on that host uses some other port. Under Windows, Linux and OSX the OS environment variables `NAMESERVERHOST` and `NAMSERVERPORT` can be set to `host` and `portno`, respectively, before the system is started.

2.1. The query interface

To execute an OSQL statement `stmt` against the embedded sa.engine system use the macro:

```
(osql stmt)
```

Every OSQL statement has a value. For example, the following statement returns an object representing the name of the new type named `PERSON` in the embedded sa.engine database:

```
Lisp 1> (osql "create type Person")
#[OID 2121 "PERSON"]
```

The result of queries returning finite object streams are returned as a list of lists, where each sublist represents an element in the object stream, for example:

```
Lisp 2> (osql "1+2+3")
((6))
```

2.1.1. The callback interface to object streams

Whereas the `osql` macro can be used only for executing local OSQL statements returning single object or queries returning finite object streams, the general client API allows the execution of any OSQL statement both locally and on remote peers and edges. The object steream interface is based on calling callback Lisp functions for every element in a returned object stream. The function `map-query-stream` applies the callback function `cb` on each element returned by executing the OSQL query `q` in the peer connection `c`.

```
(map-query-stream c q cb)
```

for example:

```
Lisp 1> (defglobal _c_ (connection))
_C_
Lisp 1> (map-query-stream _c_ "iota(1,3)" #'print)
(1)
(2)
(3)
NIL
```

In the example the global variable `_c_` is bound to a connection to the local sa.engine. The function `map-query-stream` always returns the value `nil`. The tuples from the object stream are passed to the callback function as lists. Analogously, the following calls `iota` remotely on the peer named `mypeer`:

```
Lisp 1> (defglobal _mypeer_ (connection "mypeer"))
_MYPEER_
```

```
   0.015 s
   Lisp 1> (map-query-stream _mypeer_ "iota(1,3)" #'print)
   (1)
   (2)
   (3)
   NIL
```

The function using map-query-stream first creates an internal query stream on which it immediately applies the callback function. The object stream for a query `q` to peer `p` can be obtained by calling:

```
   (query-stream c q)
```

for example:

```
   Lisp 1> (defglobal _mystream_ (query-stream _mypeer_ "iota(1,3)"))
   _MYSTREAM_
```

To apply a callback function `cb` on each element of an object stream `s`, call:

```
   (map-stream s cb)
```

for example:

```
   Lisp 1> (map-stream _mystream_ #'print)
   (1)
   (2)
   (3)
   NIL
```

Notice that not only pure queries can be executed using map-query-stream, but any statement, e.g.:

```
   Lisp 1> (map-query-stream _mypeer_ "create type Person" #'print)
   (#[OID 2121 "PERSON"])
```

Here a type named `PERSON` is created in the peer named `MYPEER`.

2.1.2. Immediately running remote queries

Queries `q` returning finite streams can be completely executed immediately by calling:

```
   (run-query-stream c q)
```

examples:

```
   Lisp 1> (run-query-stream _mypeer_ "iota(1,3)")
   3
   Lisp 1> (run-query-stream _mypeer_ "siota(1,3)")
   3
   Lisp 1> (run-query-stream _mypeer_ "1+3")
   4
   Lisp 1> (run-query-stream _mypeer_ "[1,2,3]+1")
   #(2 3 4)
   Lisp 1> (run-query-stream _mypeer_ "create type Department")
   #[OID 2127 "DEPARTMENT"]
```

Notice that the function `run-query-stream` returns the *last* element of a stream.
Notice also that bags are treated as finite streams so the last element of a bag is returned.
Notice that the results of queries returning a single values are treated as finite streams having a single element and that OSQL vectors are represented as Lisp vectors.

Notice that any OSQL statement can be executed remotely by `run-query-stream`.

2.1.3. Materialized object streams

It is often practical to convert a finite object stream s to a list of tuples by calling:

```
(object-stream-list s)
```

example:

```
Lisp 3> (object-stream-list (query-stream _mypeer_ "iota(1,3)"))
((1) (2) (3))
```

2.2. The function interface

The time to dynamically compile and optimize a query by sa.engine can be rather long, so a better way is to directly call OSQL functions using the function:

```
(call-stream c fn argl)
```

Here `fn` is the name of an OSQL function to call in connection `c` and `argl` are the actual arguments in the call represented as a list or a vector. For example:

```
Lisp 1> (defglobal _s2_ (call-stream _mypeer_ 'iota '(1 3)))
_S2_
Lisp 1> (map-stream _s2_ 'print)
(1)
(2)
(3)
NIL
```

For the common case that a called function `fn` is run immediately there is a combined call:

```
(map-call-stream c fn argl cb)
```

example:

```
Lisp 3> (map-call-stream _mypeer_ 'iota '(1 3) #'print)
(1)
(2)
(3)
NIL
```

Analogous to `run-query-stream` the OSQL function `fn` can be called immediately for arguments `argl` in connection `c` by calling:

```
(run-call-stream c fn argl)
```

example:

```
Lisp 3> (run-call-stream _mypeer_ 'sqrt '(4))
2.0
```

2.2.1. Type resolution

Since generic OSQL functions can be overloaded on different types of arguments `call-stream` and `run-call-stream` will dynamically select the OSQL object representing the applicable *resolvent* for the argument list. For example, the generic OSQL function `range()` has two resolvents with signatures:

```
range(Number u)->Bag of Integer
```

`range(Number l,Number u)->Bag of Integer`.

The 1st resolvent is used for the argument list `(2)` while the 2nd resolvent is correct for the argument list `(1 3)`. The argument list `("a")` is invalid for both resolvents:

```
Lisp 1> (map-call-stream _mypeer_ 'range '(2) #'print)
(1)
(2)
NIL
Lisp 1> (map-call-stream _mypeer_ 'range '(2 3) #'print)
(2)
(3)
NIL
Lisp 1> (map-call-stream _mypeer_ 'range '("a") #'print)
2020-11-16T19:21:22.829 Illegal function call: "range(Charstring)
Function range defined for signatures:
range(Number u)->Bag of Integer
range(Number l,Number u)->Bag of Integer"
```

The process of selecting the correct resolvent for a given generic function is called *type resolution*. The overhead of type resolution in `call-stream` and `map-call-stream` can be avoided by first calling the Lisp function `(resolve-call fn types)` that returns the object representing the correct resolvent for applying the generic OSQL function `fn` on an argument list having types named `types`, and then using the resolvent as function argument in subsequent calls to `map-stream`, for example:

```
Lisp > (defglobal _rno_ (resolve-call _mypeer_ 'range '(number number)))
_rno_
Lisp > (map-call-stream _mypeer_ _rno_ '(2 3) #'print)
2
3
NIL
```

In this case the call to `map-call-stream` is faster than in the first case where the name of generic function was given as first argument. Using `resolve-call` is favourable if `map-call-stream` is called several times.

2.2.1. Indefinite object streams

There is no upper limit on how many tuples can be retrieved from an object stream. The system is able to handle object streams containing indefinite numbers of elements, in which case calls to `map-stream` etc. will never terminate. You must then terminate the mapping by using CTRL-C. For example, try:

```
(map-query-stream _mypeer_ "heartbeat(0.5)" #'print)
```

3. Defining foreign Lisp functions

Foreign OSQL functions implemented in Lisp iteratively emits tuples in their result object streams for given arguments. This chapter describes through examples the different kinds of foreign Lisp functions.

3.1. Simple foreign functions

The following sa.engine commands implements the foreign Lisp function *hello() -> Charstring* that returns the string "Hello World":

```
[sa.engine] 1> create function hello() -> Charstring as foreign 'hello+';
#[OID 2043 "HELLO->CHARSTRING"]
[sa.engine] 2> lisp;
Lisp 2 > (defun hello+ (ff res) (osql-result “Hello World”))
HELLO+
Lisp 2> :osql
[sa.engine] 2> hello();
"Hello World"
[sa.engine] 2>
```

The foreign Lisp function implemention `HELLO+` has no argument and emits a simple object stream element containing the string “Hello World” by calling the function `osql-result` once. The first argument `ff` of a foreign Lisp function is bound to the resolvent being defined, here `#[OID 2043 "HELLO->CHARSTRING"]`.

3.2. Foreign functions returning tuples

Foreign functions can return a tuple of more than one value by passing more arguments to `osql-result`. For example, the function

```
create function sqrt2(Number x) -> (Number pos, Number neg)
  as foreign 'sqrt2-++';
```

returns both the positive and negative square roots of number `x`. It is implemented in Lisp as:

```
((defun sqrt2-++ (ff x pos neg)
  (cond ((= x 0)(osql-result x 0 0))
        ((> x 0)(osql-result x (sqrt x)(- (sqrt x))))))
```

Notice that both the argument(s) and result(s) of the function call are emitted by `osql-result`. Thus, the number of arguments of `osql-result` must be equal to the arity plus the width of the signature of the foreign OSQL function, in this case it has three arguments.

3.3. Foreign functions returning bags

Foreign functions can return bags of values. For example, the foreign function

```
create function myrange(Number n) -> Bag of Number
  as foreign 'myrange-+';
```

returns a bag of the numbers from 1 to `n`. It is implemented as:

```
(defun myrange-+ (ff n res)
  (dotimes (i n) (osql-result n (1+ i))))
```

Here `osql-result` is called once per tuple emitted to the result object stream,

3.4. Foreign functions returning streams

The same mechanism as for bags can be used for returning (possibly infinite) streams of values. For example, the function

```
create function natural_numbers() -> Stream of Integer
  as foreign 'natural-numbers+';
```

returns an infinite stream of the natural numbers (integers from 1 and up). It is implemented in Lisp as:

```
(defun natural-numbers+ (ff res)
  (let ((nn 0))(loop (osql-result (incf nn)))))
```

If you call `natural_numbers()` from the console REPL an infinite stream of numbers is returned and the system will print natural numbers until you interrupt it with CTRL-C. The call `section(natural_numbers(), 10, 20)` will return a finite stream.

3.5. Foreign functions returning vectors

For example, the function

```
create function vectors(Number n) -> Bag of Vector
  as foreign "vectors-+";
```

will return a bag of the vectors `[0]`, `[1]`…`[n]`. It is implemented in Lisp as:

```
(defun vectors-+ (ff n res)
  (dotimes (i n)(osql-result n (vector (1+ i)))))
```

3.6. Foreign functions returning records

For example, the function

```
create function records(Number n) -> Bag of Record
  as foreign "records-+";
```

will return a bag of the records
`{"k":1,"v":"1"},… {"k":n,"v":"n"}.`

It is implemented in Lisp as:

```
(defun records-+ (ff n res)
  (dotimes (i n)
    (osql-result n (make-record (vector (mkstring (1+ i)) (1+ i))))))
```

3.7. Foreign aggregate functions

Aggregate functions such as `sum(Bag of Number b) -> Number` compute a single object for a given collection `b`. Foreign aggregate functions are implemented by using `mapstream-apply` to map over the collection `b`.

For example, the function

```
create function mysum(Bag of Number b) -> Bag of Number
  as foreign "mysum-+";
```
will sum up all numbers in bag `b`. It is implemented in Lisp as:
```
(defun mysum-+ (ff b res)
  (let ((sum 0))
    (mapstream-apply b #'(lambda (i) (incf sum i)))
    (osql-result b sum)))
```
If you call `mysum(range(1,10))` from the console REPL you will get the result 55. The function `(mapstream-apply s fn)` applies the Lisp function `fn` on each element of the stream, bag, or vector `s`.

3.8. Streamed aggregate functions

A regular aggregate function that produces its result by mapping over an entire collection cannot be applied on indefinite object streams. By contrast a *streamed aggregate function* produces the elements of a result stream by computing accumulated stream elements for each element of an input stream. For example, the built-in function `rsum(Stream of Number s) -> Stream of Number` computes the *running sum* of the elements in `s`. Streamed aggregate functions are implemented in Lisp by calling `osql-result` in the callback function. For example, the function
```
create function myrsum(Stream of Number s) -> Stream of Number
  as foreign "myrsum-+";
```
that calculates the running sum for each element in bag `s` is implemented in Lisp as:
```
(defun myrsum-+ (ff s res)
  (let ((sum 0))
    (mapstream-apply s #'(lambda (x)
                            (incf sum x)
                            (osql-result s sum)))))
```

3.9. Multi-directional foreign functions

Foreign OSQL functions can be made invertible. For example, assume a foreign square root function:
```
create function sqroot(Number x) -> Bag of (Number r)
  as foreign 'sqroot-+';
```
Its definition in Lisp is:
```
(defun sqroot-+ (ff x y)
  (cond ((= x 0)(osql-result x 0))
        ((> x 0)
         (osql-result x (sqrt x))
         (osql-result x (- (sqrt x))))))
```

The bag of numbers 2.0 and -2.0 will be returned by the query `sqroot(4)`.

If `sqroot` were invertible we could also make a query calling its inverse:
```
select x from Number x where 2 in sqroot(x); /* Result is 4 */
```
Since the definition above is *not* multidirectional the system will raise an error that the query is not executable because variable `x` is not bound.

We now define `sqroot` as an invertible foreign Java function by redefining it as:

```
create function sqroot(Number x) -> Bag of (Number r)
  as multidirectional ('bf' foreign 'sqroot-+')
                      ('fb' foreign 'sqroot+-');
```

where sqroot+- is defined as:

```
(defun sqroot+- (ff x y)
  (osql-result (* y y) y))
```

The method `sqroot+-` implements the inverse of `sqroot-+`.

References

1. Tore Risch: *aLisp User's Guide*, Stream Analyze Sweden AB