LEMMA Interface Definition

David C.J. Matthews and Thierry Le Sergent

Abstract

Implementing a functional language on a parallel machine, especially on a loosely coupled machine such as a network of workstations, is a hard task. Most of the problems are, however, language independent particularly the memory management. For this purpose, we have designed and implemented LEMMA, a "memory platform" which allows the upper layer of the distributed implementation to access in a straightforward manner objects allocated by another machine, and reclaims automatically allocated space which is no longer used by any machine.

This report is a detailed description of the interface provided by LEMMA. It is in fact a reference manual for people who wish to implement a run-time system for a language on top of it. It does not describe the way LEMMA is implemented, leaving open the possibility of other implementations on different concurrency platforms providing the same interface.

1 Introduction

Our original aim was to build a distributed implementation of the languages Poly/ML[1], developed at the University of Edinburgh (GB), and LCS[2], designed at LAAS-CNRS, Toulouse (France). The design of the distributed implementation arose out the need to produce a common platform for both Poly/ML and LCS. Between us we had experience of garbage-collectors and distributed memory and we wanted to put this together. We had also worked on different languages and did not want to commit to using one or the other. The design had to support two languages with different ways of representing data and very different views of processes and communication at the language-level. By support here, we mean the need to be able to be configured so that it can be incorporated into the run-time system for the language, not interoperability.

---

[1] David C.J. Matthews, "A Distributed Concurrent Implementation of Standard ML". Proc. EurOpen Autumn 1991 conference, Budapest. Reprinted as LFCS Report ECS-LFCS-91-174, Department of Computer Science, University of Edinburgh.

[2] Bernard Berthomieu and Thierry Le Sergent, "Programming with behaviors in an ML framework: the syntax and semantics of LCS", in Programming Languages and Systems - ESOP '94, LNCS 788, April 1994.

This definition gives the interface to a particular implementation of a memory system for ML-like languages. As far as possible the interface has been designed to be independent of the details of the language implemented on it and also as independent as possible of the memory and communications system on which it is implemented. The present implementation runs on a network of workstations connected by a high-speed network. It would be equally possible to provide an implementation of the interface on a more closely coupled parallel machine.

1.1 Terminology: RTS and LEMMA

Many of the functions of a garbage collector involve scanning objects to find the addresses and then processing those addresses. Different languages have different ways of encoding information such the size of an object and how pointers and numbers are distinguished. To allow our garbage collector to be independent of this object representation we need to separate out the parts which encode this information.

We call the shared memory system and the language-independent part of the garbage-collector, LEMMA. The choice of LEMMA as a name for our system comes from the idea that, as a lemma is often the essential part of the proof of a theorem, so LEMMA is the central part of the distributed implementations. LEMMA may also stand for "Laas and Edinburgh University Ml Memory Architecture". The rest of the run-time system is known as the run-time system (RTS).

To some extent LEMMA can be thought of as a library of functions which are called by the RTS. For example, there is a function `LEMMA_allocate_immutable` which allocates space in rather the same way as the Unix malloc function. However, if there is insufficient store available LEMMA will need to start a garbage collection. To do that it needs to get information from the RTS. Most obviously, it needs to find the root pointers which it does by calling `RTS_apply_to_roots`. Note that LEMMA functions all begin with `LEMMA_` and the RTS functions by `RTS_`. In this way LEMMA treats the RTS rather like a library.

The security of LEMMA is as dependent on the RTS functions behaving properly as the RTS is dependent on LEMMA. LEMMA does not test all the conditions on which it relies (unlike an operating system which must ensure its own security).

1.2 Terminology: Client and Servers

LEMMA can be used to provide an efficient asynchronous garbage collector on a single machine, but that is not its main purpose. The main aim is to provide support for running parallel applications on a group of machines.

The model of computation assumed in this definition is of a single **client** machine which probably will provide an interactive session, and one or more **servers**. The servers provide only computing power. A server maintains state only as an extension of the client, and when the client exits, the servers exit as well.

1.3 Overview

The main body of this report describes in detail each function of LEMMA and the RTS. Of course, the roles and constraints on each of them are interdependent. In order to ease the understanding of the explanations given in the next section, an overview of LEMMA and its relationship with the RTS, are presented here.

LEMMA has at least two roles:

- To give to the RTS a way to make connections with servers, and to allow these servers and the client to communicate by asynchronous messages (sections 7 and 8).

- To allow the RTS to allocate memory without requiring explicit deallocation, and to allow any machine to use an object allocated by another machine knowing only the address of the object.

LEMMA is designed to be efficient for supporting parallel implementation of languages of the ML type, of which Poly/ML and LCS are two. To achieve this efficiency we have had to make some assumptions. The most important one is that the majority of the objects created by the RTS are *immutable*, by which we mean that the value given at the time of allocation is never overwritten, and the fact that an object is immutable is known at the allocation time. The consequence is that mutable and immutable objects are handled in completely different way:s

- Mutable objects must be accessed by the RTS through a set of functions (section 6).

- Immutable objects can be accessed (read) directly from their address. That may cause a system trap when the object has been allocated by another machine. The fault can be handled by LEMMA which includes a distributed shared virtual memory mechanism. But it is the RTS which will receive the signal; it will have to call a LEMMA function (see section 3). The reason is that an RTS may use these traps for other purposes. Similarly, LEMMA may send IO signal to the RTS which has to call back a LEMMA function.

To reuse the space occupied by objects that are no longer reachable the current implementation of LEMMA uses an incremental two-space copying garbage collector. This has several consequences: LEMMA has to assume that the RTS is able to distinguish addresses of objects stored within the objects themselves (section 2). It needs to ask the RTS for the roots of all objects handled by the RTS, for the length of objects, etc. The RTS has to provide to LEMMA a set of functions which in effect encapsulate the structure and encoding of the objects (section 4).

One of the difficulties of having LEMMA acting as a coroutine with the RTS, is that the points of the execution of the RTS where LEMMA can act must be made clear. The most important constraints are that an RTS must fill the content of an object before calling another LEMMA function. There is in addition a mechanism of asynchronous interruption of the RTS. It is used for example when one machine, by asking for memory space, causes a global garbage collection to start. All the machines have to synchronise in that case, so LEMMA has to interrupt the RTS's, but they cannot be stopped at any point of their computation. The same problem arises for the messages exchanged between the RTS's. In all these cases LEMMA notifies the RTS that something happened, and the RTS must call back LEMMA as soon as possible. See section 9 for the details.

Several optimisations have been introduced in LEMMA.

- Allocation of immutable data is very frequent. To speed up these allocations, the RTS is allowed, under certain constraints, to allocate a segment of memory, and then put several objects in it. See section 5 for the details.

- In some cases LEMMA may prefer to copy objects between the machines instead of using its internal distributed shared virtual mechanism. This makes use of the fact that ML does not allow the user to distinguish between two pointers to the same immutable data structure, or two different data structures which happen to have the same values throughout. There may be cases when the RTS would prefer that LEMMA does not use this optimisation, for example if it uses the addresses of the objects for an equality function.

- One interesting feature of LEMMA is that most of the functions it provides can be called asynchronously, by which we mean that the RTS will not be blocked waiting for the result if the execution of the LEMMA function involves a dialogue between several machines (sections 2, 3, 6 and 9). Instead, the RTS may switch to another task. That is the case in the Poly/ML and LCS systems where each RTS includes a scheduler to run several Poly/ML or LCS processes.

1.4 Using LEMMA

LEMMA is quite a large system which is capable of supporting a range of languages with different run-time systems. Many of the features are only required in special circumstances and it is possible to write a simple run-time system using only the basic facilities. For example, a simple system could always call `LEMMA_ensure_object_is_accessible` synchronously, thus avoiding the need for the scheduler to suspend processes which are waiting for objects on other machines. Many run-time systems will not need to freeze objects or to use weak garbage-collection roots.

To use LEMMA it is necessary to define all the RTS functions. Since there are eighteen of them this may seem like a lot of work. Most, however, will be simple calls to other functions or tests on the value passed as argument.

2 Standard Types

`LEMMA_word`, `LEMMA_addr`, `RTS_word` and **`RTS_addr`.**

```
typedef int LEMMA_word, *LEMMA_addr;
typedef int *RTS_addr;
```

`LEMMA_word` is the basic unit of the memory and `LEMMA_addr` a pointer to a `LEMMA_word`. A `LEMMA_word` will normally be a machine word and will be large enough to contain a `LEMMA_addr`. A `LEMMA_addr` always points to the start of an **object**, which consists of one or more whole LEMMA words.

The easiest way to implement a run-time system using LEMMA is for the run-time system to store LEMMA addresses and to use LEMMA addresses internally. Most run-time systems will work in this way. However, LEMMA allows the RTS to choose an alternative representation of an address, an RTS address, and to provide functions or macros to convert between RTS addresses and LEMMA addresses. The reasons why this might be useful are for backwards compatibility or to allow for pointers inside objects. For example Poly/ML uses RTS addresses which point at the second word of an object, and a special encoding of return addresses which may point inside a piece of code. LEMMA makes the restriction that a run-time system address always points at the first word of the object (i.e. the same as a LEMMA address) or at some byte within the object. `RTS_addr` is used for the type of addresses used within the run-time system. When LEMMA requires the LEMMA address for an object described by an RTS address it will use `RTS_start_of_object` to do the conversion.

LEMMA assumes that objects are self-describing so that, given an address of an object, it is possible to identify the values within it which are addresses. It does not actually require that the values themselves are tagged to distinguish between numbers and addresses, although that is probably the easiest implementation.

LEMMA_synch_type

```
typedef enum {
   LEMMA_SYNCH,
   LEMMA_ASYNCH
} LEMMA_synch_type;
```

This type is used as an argument for a number of functions. It is used for functions which may take some time to complete. The call can be made synchronously (`LEMMA_SYNCH`) in which case the function does not return until the operation is complete and the result, if any, has been delivered. An asynchronous request (`LEMMA_ASYNCH`) causes the request to be started but the function does wait for it to complete. Instead the result can be checked later. The process scheduler in the run-time system can make use of this to schedule another process.

LEMMA_result

```
typedef enum {
    LEMMA_Failed,
    LEMMA_Succeeded,
    LEMMA_InProgress,
    LEMMA_TooMany
} LEMMA_result;
```

`LEMMA_result` is used as the result of most LEMMA functions. `LEMMA_Succeeded` is the usual result and indicates that the operation succeeded. `LEMMA_Failed` indicates a failure of some sort, usually invalid arguments. `LEMMA_InProgress` is returned as the result of an asynchronous request when the operation has been started successfully but has not yet completed. `LEMMA_TooMany` is returned if an asynchronous request is made and for some reason, usually insufficient buffer space, LEMMA is unable to start the request at the moment. This is not a failure and indicates a temporary situation. The RTS should try the operation again later.

In the descriptions of functions below many of the functions only return `LEMMA_Succeeded` or `LEMMA_Failed`. In that case the return values are not given if the function returns `LEMMA_Succeeded` in the normal case and `LEMMA_Failed` only if the arguments are invalid.

3 Signal handling

LEMMA uses some operations which requires the RTS to set up signal handlers. Because the RTS may use these signals for its own purposes LEMMA does not set up its own handlers and relies on the RTS to do that and to call the appropriate functions.

LEMMA_ensure_object_is_accessible

```
LEMMA_result LEMMA_ensure_object_is_accessible(RTS_addr
    addr, LEMMA_synch_type synch);
```

Store may be protected as part of the shared memory system or the garbage collection. Accessing such an object will result in a segmentation violation (SIGSEGV) or a bus error (SIGBUS) depending on the version of the operating system or the actual address. The RTS must call `LEMMA_ensure_object_is_accessible` to handle the fault.

The `addr` argument is an address which either points to the start of the object or at some byte inside it. Although its type is given as `RTS_addr` it will not be passed to `RTS_start_of_object`. The faulting address can usually be obtained from the signal handler and this can be passed to `LEMMA_ensure_object_is_accessible`.

The `synch` argument indicates whether the request is synchronous or asynchronous. Synchronous requests cause the function to block until the request has completed. Asynchronous requests may be completed immediately and return `LEMMA_Succeeded` if they will not result in a delay, otherwise the request is started and the function returns `LEMMA_InProgress`. `LEMMA_TooMany` may be returned if LEMMA cannot process the result at the moment. There is no function to check whether the operation has succeeded. When an operation completes the RTS will be informed by a call to `RTS_Interrupt`. The instruction which caused the fault should be retried and if the operation has succeeded the access will succeed.

Return values

`LEMMA_Succeeded`
: The operation has completed. The object is readable. Note that there are no guarantees about the time for which the object will remain readable and it is possible that by the time control returns to the caller the object may no longer be accessible.

`LEMMA_Failed`
: The operation has failed because the address was invalid.

`LEMMA_InProgress`
: An asynchronous operation is in progress.

`LEMMA_TooMany`
: An asynchronous request was made but there are currently too many requests outstanding The request should be retried later.

LEMMA_handle_IO

```
void LEMMA_handle_IO(void);
```

This function should be called when a SIGIO (SIGPOLL) signal is received. It is used to process communications. Because signals can arrive at any time there are restrictions on the operations that LEMMA will perform as the result of this call. In particular it will not start a garbage collection. It may call `RTS_Interrupt` if it wishes to notify the RTS of a particular event.

4 Garbage collection

Strong and Weak Roots

The function of a garbage collector is to give the illusion of an infinite amount of memory on a system which has only a limited amount of memory, by reclaiming space that is no longer accessible. There are often other resources, such as file descriptors, which are in short supply and LEMMA provides a facility which helps them to be managed in a similar way. This is done by associating memory objects with the resources and then reclaiming the resources if the associated memory object is reclaimed. So, for example, when a file is opened a **token** is created and handed back to the program. This represents the file and may contain other information as well. Whenever the program wishes to access the file it presents the token. If the file is explicitly closed the token may be overwritten. Alternatively an entry in a table maintained by the run-time system may be overwritten. If the file is not explicitly closed, but simply "dropped", the token will become unreachable and may become garbage. The file can then be closed.

There is no mechanism for the garbage-collector to explicitly indicate to the RTS that any object has been deleted, and it is difficult to see how such a mechanism could be provided. Instead, the RTS is expected to keep a table of the tokens it has given out. These are known as **weak roots**, because they are roots for the garbage-collector, but do not retain the object they refer to unless the object is also reachable from another **strong** root. LEMMA will call `RTS_apply_to_roots` with the `strength` argument set to `LEMMA_GC_Kind_Weak` and the RTS should apply the `op` argument to each of its weak pointers. If there is no other reference to the pointer the `op` function will return zero. The RTS should then release the associated resource.

LEMMA_force_garbage_collection

```
LEMMA_result LEMMA_force_garbage_collection(void);
```

Forces the start of a garbage collection. This is probably most useful in the case when a resource controlled by weak pointers has been exhausted and needs to be recovered.

LEMMA_update_weak_pointer

```
RTS_addr LEMMA_update_weak_pointer(RTS_addr);
```

This must be called whenever a weak pointer is accessed. It ensures that the weak pointer refers to a reachable object if the garbage-collector is active. The argument is the current value of the pointer. The result will normally be the same as the argument except when the object has been moved by the garbage collector. The result should be stored in place of the old pointer.

LEMMA_synchronise

```
void LEMMA_synchronise(void);
```

This function must be called by the RTS as soon as possible after LEMMA has called `RTS_Interrupt` with reason code `LEMMA_Synch_request`. It may result in a garbage collection so it must not be called directly from `RTS_Interrupt` unless the RTS is certain that any objects in the process of being allocated have been correctly initialised.

Run-time system routines

RTS_apply_to_roots

```
typedef enum {
    LEMMA_GC_Kind_Strong,
    LEMMA_GC_Kind_Weak
} LEMMA_GC_Kind;

void RTS_apply_to_roots(RTS_addr (*op)(RTS_addr),
     LEMMA_GC_Kind strength);
```

This function is called to process all the root addresses, that is all the addresses which the RTS uses to access data in the memory. This function should apply `op` to these addresses. If `strength` is `LEMMA_GC_Kind_Strong` it should be applied to the normal roots and if it is `LEMMA_GC_Kind_Weak` it should be applied to the weak roots (if any). In that case the `op` may return 0 if the weak reference is the only one which refers to the object.

RTS_for_addresses_in_object

```
void RTS_for_addresses_in_object(LEMMA_addr base,
     void (*op)(RTS_addr, LEMMA_addr, int))
```

LEMMA will call this function to perform one of a number of operations on the addresses in an object. Since the layout of objects is under the control of the RTS this function is needed. The RTS should call the `op` argument with each of the RTS addresses in the object, the `base` value passed to it and the offset within the object where this address was found. If LEMMA needs to change an address it will call RTS_update_address with a new RTS address and the `base` and `offset` values. Although `offset` is intended to be the offset of the affected word in the object, it is not interpreted by LEMMA and is only ever passed to RTS_update_address. The RTS can therefore use it to encode information if required.

RTS_for_addresses_in_object may read any word in the object simply by loading the value. It must not use LEMMA_load_mutable if the object is mutable. It must not read any other object in the memory, and specifically must not try to access any of the objects referred to by the addresses it passes to `op` .

RTS_update_address

```
void RTS_update_address(RTS_addr newaddr, LEMMA_addr base,
     int offset)
```

This function is called by LEMMA in order to update an address in an object. The arguments are the new address in `newaddr`, the address of the of the object in `base` and the `offset`. `offset` will be a value originally passed to the `op` argument of a call to `RTS_for_addresses_in_object`. This function may be called recursively from `op` or LEMMA may store the information and call this function later.

This function must not read or write any other object in the memory, and specifically must not try to access the object pointed at by `newaddr`.

RTS_start_of_object

```
LEMMA_addr RTS_start_of_object(RTS_addr p);
```

It should return LEMMA address for an RTS address of `p`. If RTS addresses always point to the start of the object, this function will simply return its argument. This function may read the object in order to do the conversion so it is only ever called by LEMMA when the object referred to is readable.

RTS_object_length

```
int RTS_object_length(LEMMA_addr p);
```

Returns the number of words used by an object with given LEMMA address.

RTS_fill_unused_space

```
void RTS_fill_unused_space(LEMMA_addr p, int w);
```

LEMMA assumes that pages are completely filled with objects. It calls this function to place a dummy object at address `p` which will occupy exactly `w` words. A dummy object may, for example, be a segment marked as uninterpreted bytes, or with constants. If appropriate the RTS may use several objects to fill up the space.

RTS_copy_an_object

```
void RTS_copy_an_object(LEMMA_addr oldaddr,
     LEMMA_addr newaddr, int length);
```

This function is called when an object must be copied from one address to another. It moves the object with LEMMA address `oldaddr` and occupying `length` words to the new address `newaddr`. In almost all cases it will simply be a byte-copy operation but if the object contains internal addresses which cannot be processed by functions such as `RTS_for_addresses_in_object` then these should be adjusted here. In Poly/ML that occurs with stacks which contain pointers within the stack.

RTS_adjust_addr

```
RTS_addr RTS_adjust_addr(RTS_addr original,
     LEMMA_addr oldbase, LEMMA_addr newbase);
```

When LEMMA address of an object has been moved from `oldbase` to `newbase`, this function is called to compute the new RTS address. If RTS addresses are always the same as LEMMA addresses `original` and `oldbase` will be the same and this function can simply return `newbase`.

RTS_can_duplicate

```
int RTS_can_duplicate(RTS_addr addr);
```

This function is called by LEMMA when it wishes to make a duplicate copy of an immutable object. This may be used when objects are shared between machines and it may be more efficient to duplicate the objects on the two machines rather than to maintain the sharing. The semantics of equality in ML for immutable data mean that they will continue to be equal even though they are at different addresses. There may be cases, however, when the RTS wishes to maintain the identity of immutable data. In that case this function may return zero if the object passed as argument must not be duplicated but that may result in a performance penalty. Mutable objects are never duplicated.

Forwarding Pointers

Forwarding pointers are used by the garbage-collector to indicate the address of an object after it has been moved. It is necessary to place the forwarding pointer in the old object and the RTS is responsible for storing and retrieving the address and indicating whether an object contains a forwarding pointer. The forwarding pointer may overwrite the contents of the object because, if an object contains a forwarding pointer, no access will be made to it other than to extract the forwarding pointer. In particular, `RTS_object_length` will not be called, so if the object contains a field with the length information, this can be used to encode the presence of a forwarding pointer, by, for example, setting the length to zero or a negative number.

RTS_put_forwarding_ptr

```
void RTS_put_forwarding_ptr(LEMMA_addr old , LEMMA_addr
     new);
```

LEMMA calls this function to place an address `new` in an object at address `old`.

RTS_contains_forwarding_ptr

```
int RTS_contains_forwarding_ptr(LEMMA_addr p);
```

Returns non-zero if the object with LEMMA address `p` contains a forwarding pointer.

RTS_get_forwarding_ptr

```
LEMMA_addr RTS_get_forwarding_ptr(LEMMA_addr p);
```

Extracts a forwarding pointer from an object. Only called if `RTS_contains_forwarding_ptr(p)` has returned non-zero.

5 Immutables

Most objects in ML programs are created and initialised and thereafter their contents remain unchanged. Such objects are referred to as **immutable**, in contrast with **mutable** objects such as references and arrays. The type system ensures this and so the implementation can rely on it. In particular an implementation can tell when an object is created whether it is immutable or mutable. This is in contrast with languages such as C where, although the contents of a particular object may be changed much less frequently than it is read, there is no restriction which prevents a piece of store from being changed.

LEMMA makes use of the distinction between mutable and immutable objects to provide a number of optimisations and assumes that the majority of objects will be immutable, and that the mutable objects may well be updated frequently. It is possible to use LEMMA to implement a language where all objects are mutable, but it is likely to be less efficient than an implementation built with the assumption that all objects are potentially updatable, but that any particular object has a low chance of being updated.

LEMMA assumes that after the store for an object has been allocated the object is fully initialised before LEMMA is called again. The only exception is that the interrupt handling functions, `LEMMA_handle_IO` and `LEMMA_ensure_object_is_accessible` may be called. In the case of a mutable object the store can be given initial values and changed later, but immutable objects must be given the values they will retain for the whole of their lifetimes. The RTS should not rely on LEMMA initialising the memory to any particular value when an object is allocated.

LEMMA_allocate_immutable

```
LEMMA_result LEMMA_allocate_immutable(LEMMA_addr *addr,
        int size)
```
Allocates an individual immutable object. The object must be completely filled in before the next call to any allocation routine. Hence it cannot form part of a circular structure which does not involve a mutable object (except the trivial case when the object contains its own address). The `size` argument gives the number of words required. The address of the newly allocated area is returned by storing it at the address given in the `addr` argument.

LEMMA_allocate_immutable_segment

```
LEMMA_result LEMMA_allocate_immutable_segment(LEMMA_addr
        *base, int *size)
```
Allocates a segment to contain immutable objects. There may only be one immutable segment active at a time. The RTS must fully fill each immutable segment before starting a new one, inserting a dummy object if necessary (see RTS_fill_unused_space). The value pointed at by the `size` argument should be set to the minimum number of words required. On return it will be set to the number of words actually available. This will be at least the size required.

The segment may be used to allocate many objects. Each object must be fully initialised before a LEMMA function (other than `LEMMA_handle_IO` or `LEMMA_ensure_object_is_accessible`) is called. A call to such a LEMMA function may result in LEMMA calling `RTS_close_segment` to stop further allocation in the segment so the RTS must not rely on being able to continue allocating in the same segment.

Note that the objects in a segment may form circular structures.

Run-time system routines

`RTS_close_segment`

```
void RTS_close_segment()
```

LEMMA may call this function to prevent further allocation in an immutable segment. When this function is called the RTS must not allocate any more store in the allocation segment and it must fill any unused space with a dummy object (see `RTS_fill_unused_space`). This function should not make calls to any LEMMA function.

`RTS_close_segment` is only called as a result of a call to a LEMMA function which is not an interrupt handler (i.e. it will not be called by `LEMMA_handle_IO` or `LEMMA_ensure_object_is_accessible`). Since the RTS is only allowed to call such LEMMA functions when any allocated objects have been fully initialised this function will not be called by LEMMA unless all allocated objects have been initialised.

6 Mutables

`LEMMA_allocate_mutable`

```
LEMMA_result LEMMA_allocate_mutable(LEMMA_addr *addr,
        int size);
```

Allocates space for a mutable object. The size is the number of words for the object. LEMMA may add extra information to the object but this is transparent to the RTS. The address of the start of the storage is returned by assigning the address to the `addr` variable. A mutable must be completely initialised before any other LEMMA function is called (other than one of the interrupt handlers).

`LEMMA_load_mutable` and **`LEMMA_assign_mutable`**

```
LEMMA_result LEMMA_load_mutable(int is_address,
        LEMMA_addr addr, int offset, LEMMA_word *result,
        LEMMA_synch_type synch, int *synchtoken);

LEMMA_result LEMMA_assign_mutable(int is_address,
        LEMMA_addr addr, int offset, LEMMA_word newvalue,
        LEMMA_word *result, LEMMA_synch_type synch,
        int *synchtoken);
```

`LEMMA_load_mutable` loads the current value of a word from a mutable object, and `LEMMA_assign_mutable` sets a word and optionally returns the previous value. If the previous value is required the operation is guaranteed to be atomic.

Most of the arguments are the same for both functions. The address `addr` is the base of the object as returned when it was created. The `offset` is the number of the word within the object which is to be read or written. It must be between zero and one less than the size of the object. For `LEMMA_assign_mutable` the value to be put in the word is passed in `newvalue`.

The request may be made synchronously or asynchronously depending on the `synch` argument. If this argument is `LEMMA_SYNCH` then the function does not return until the operation as completed and the result is available. In this case the result is placed at the address pointed at by `result`. `result` may be NULL for LEMMA_assign_mutable if the previous value of the particular word is not required. The `synchtoken` argument is not used and may be NULL.

If the request is made asynchronously (`synch` is `LEMMA_ASYNCH`) the function may return the result immediately as with the synchronous case, or the request may be started and the integer pointed at by `synchtoken` will be set to a token which can be passed to `LEMMA_check_mutable_operation`.

The `is_address` argument is used in connection with the `RTS_apply_to_address` function. It is sometimes necessary for LEMMA to process the values being assigned or being returned before they have been placed in the destination object or returned to the RTS. For example, a garbage collection may occur. The `is_address` argument should normally be set to a non-zero value indicating that the value may be an address and can be passed to `RTS_apply_to_address`. It should only be set to zero if the word to be affected contains data which cannot be an address. More specifically, it may be zero if the value being assigned or returned cannot safely be passed to `RTS_apply_to_address`. This might occur if the object was a string and the value being assigned or read consisted of bytes.

Return values

`LEMMA_Succeeded`
: The operation has completed. If `result` is not NULL the result will have been placed at the appropriate address.

`LEMMA_Failed`
: The operation has failed probably because one of the arguments was invalid.

`LEMMA_InProgress`
: An asynchronous operation has been started and is progress. A token will have been placed in the address contained in `synchtoken` .

`LEMMA_TooMany`
: An asynchronous request was made but there are currently too many requests outstanding The request should be retried later.

LEMMA_check_mutable_operation

```
LEMMA_result LEMMA_check_mutable_operation(int synchtoken,
      LEMMA_word *result);
```

Used to check whether an asynchronous load or assign has completed, and to obtain the result if one is needed. It must always be called even for an assignment when no result is required. `synchtoken` should be the token returned from `LEMMA_load_mutable` or `LEMMA_assign_mutable`. The result will be placed at the address contained in the `result` argument. This may be NULL if no result is required. The result will be undefined if the original request was a call to `LEMMA_assign_mutable` with a NULL `result` argument.

LEMMA_check_mutable_operation must be called to check for completion of every asynchronous load or assign until it returns LEMMA_Succeeded. Once the operation has succeeded it must not be called again with that token value.

Return values

LEMMA_Succeeded
: The operation has completed. If result is not NULL the result will have been placed at the appropriate address.

LEMMA_Failed
: The operation has failed probably because one of the arguments was invalid.

LEMMA_InProgress
: The operation has not yet completed and is still in progress.

LEMMA_freeze_mutable

```
LEMMA_result LEMMA_freeze_mutable(LEMMA_addr);
```

The rules on immutable objects restrict the way in which structures can be built up. Sometimes immutable data structures are needed which are more complex than can be achieved within those rules. For example, a cyclic structure cannot be produced if each object must be completely filled in before the next object is created.

To achieve these structures a mutable object can first be created and then frozen using this function. Before freezing, the object must be accessed by LEMMA_load_mutable and LEMMA_assign_mutable. LEMMA_freeze_mutable ensures that all copies of the object are coherent so that the store can be accessed directly by an machine which currently has a copy or later gets one. LEMMA_load_mutable and LEMMA_assign_mutable must not be called on the object after it has been frozen. Just before the object is frozen the RTS should mark the object so that RTS_is_mutable will return false.

LEMMA_get_exclusive_copy and LEMMA_release_exclusive_copy

```
LEMMA_result LEMMA_get_exclusive_copy(LEMMA_addr,
     LEMMA_addr*);
LEMMA_result LEMMA_release_exclusive_copy(LEMMA_addr);
```

Occasionally the overheads involved in calling LEMMA_load_mutable and LEMMA_assign_mutable are too high. LEMMA_get_exclusive_copy can be used in special cases.

LEMMA_get_exclusive_copy ensures that a local copy of the object has been obtained. The object can then be read and written directly by the machine-level load and store operations, but only on the machine which called this function. The contents of the object on any other machine is undefined, and if any other machine accesses the object before the local copy has been released, the effects are undefined. LEMMA provides no interlocking or coherency.

This function is needed for objects, such as stacks, which need to be accessed extremely efficiently, but by only one machine. In this case the RTS can ensure the interlocking itself.

A local copy is released by calling `LEMMA_release_exclusive_copy`. After this function is called another machine may get an exclusive copy or any of the machines may access the object using the `LEMMA_load_mutable` and `LEMMA_assign_mutable` functions.

LEMMA_is_mutable

```
LEMMA_result LEMMA_is_mutable(RTS_addr addr, int *result);
```

Tests whether the object is currently in a mutable area of memory. Sets the integer pointed at by the `result` argument to non-zero if `addr` is in a mutable area of memory, and to zero if it is in an immutable area. This is needed when accessing an object which may be mutable or immutable since immutable store can be read directly but mutable objects must always be read using `LEMMA_load_mutable`.

For example, the function in ML which tests for equality of arbitrary structures must use this before looking at the object itself.

Run-time system routines

RTS_apply_to_address

```
void RTS_apply_to_address(LEMMA_word *p,
      RTS_addr (*op)(RTS_addr));
```

This function is called by LEMMA to process a word which may contain an address, i.e which has been passed to `LEMMA_load_mutable` or `LEMMA_assign_mutable` with the `is_address` argument non-zero. The function should test the value pointed at by `p` and apply `op` to it if it is an address. `op` returns a possibly changed value which should be stored at the location pointed at by `p`. If the value at `p` is not an address it should do nothing.

RTS_object_is_mutable

```
int RTS_object_is_mutable(LEMMA_addr p)
```

This function is called for objects that were created as mutable, and should return non-zero if the object is still mutable. This is only of relevance if objects can be frozen, and if a run-time system never freezes objects then this function can always return non-zero.

7 Initialisation

LEMMA_initialise_memory

```
LEMMA_result LEMMA_initialise_memory(int is_server,
      int token, LEMMA_addr *interface_addr,
      unsigned interface_size);
```

Initialise LEMMA. This function must be called before any of the other LEMMA functions are called. `is_server` should be 1 if this machine has been set up as a server. The call may block until a client connects.

The token can be any arbitrary value. It is used only in a server. A connection will only succeed with a server if the same token value is used when a connection is attempted with

LEMMA_connect_to_server. It may be used, for example, to ensure that connections are only made with machines with the same architecture or servers forked by the same user.

The interface_addr and interface_size are used to create an area of memory which will be the same on the client and all the servers, and, if the database facility is used, for different runs of the program. This is useful if the memory contains machine code segments which need to make calls into the run-time system. A piece of code compiled on one machine can be copied across between the machines and call into the run-time system on that machine. It is probably not useful in an interpreted system. The interface_addr may be NULL and the interface_size zero if no interface space is required.

LEMMA_connect_to_server

```
LEMMA_result LEMMA_connect_to_server(char *server_name,
        int *server_address, int token);
```

This function must only be called on the client. It makes a connection to a new server. The name is interpreted as the name of the server. The interpretation of the string depends on the implementation of LEMMA. In the present implementation it corresponds to a machine name which is looked up using the normal host naming scheme. If the connection succeeds the function sets the value pointed at by server_address to an integer which represents the server. This value can be used directly by the client or passed to other servers to send RTS messages to this server.

Return values

LEMMA_Succeeded
: The connection succeeded.

LEMMA_Failed
: The function was called from a server, or no server could be found with the given name.

LEMMA_server_no

```
int LEMMA_server_no;
```

The address of the current machine. This may be used by other machines to send RTS messages to this machine (see section on RTS messages below). The addresses of all the machines are guaranteed to be different. This value is set by LEMMA_initialise_memory.

8 RTS Messages

RTS messages are a facility provided by LEMMA which allows the schedulers on the various machines to communicate. They consist of two words which can be used for any purpose. Typically, one word will be used as a reason code and the other word as an argument. Either of the words can be addresses. The LEMMA passes them between the machines without interpretation except through calls to RTS_apply_to_rts_message.

```
typedef struct {
    LEMMA_word rtsm_data0, rtsm_data1;
    int rtsm_sender;
} LEMMA_rts_message;
```

LEMMA_get_rts_message

```
LEMMA_result LEMMA_get_rts_message(LEMMA_rts_message *);
```

Checks to see if an RTS message has arrived, and reads it if it has. There may be limited buffering space for RTS messages so the RTS should attempt to call this function as soon as possible after receiving an interrupt with reason code `LEMMA_RTS_Msg_arrived`.

Return values

`LEMMA_Succeeded`
: An RTS message has been read and removed from the queue.

`LEMMA_Failed`
: No RTS message was waiting.

LEMMA_send_rts_message

```
LEMMA_result LEMMA_send_rts_message(int destination,
        LEMMA_rts_message *);
```

Sends an RTS message to a particular destination. There may be a limited amount of buffering available so this function may return `LEMMA_TooMany`. If buffer space becomes available LEMMA will interrupt with `LEMMA_RTS_Clear_to_send`. The RTS can then try sending again.

Return values

`LEMMA_Succeeded`
: The message has been sent. This return code does not mean that the destination RTS has actually read the message.

`LEMMA_Failed`
: The operation has failed, probably because the destination was not a valid server.

`LEMMA_TooMany`
: There were too many RTS messages which have been queued for that destination and not yet received by the RTS there.

RTS_apply_to_rts_message

```
void RTS_apply_to_rts_message(LEMMA_rts_message *rtsm,
        RTS_addr (*op)(RTS_addr));
```

Similar to `RTS_apply_to_address`. If the RTS message contains an address `op` should be applied to the value and the result used to replace the value in the message.

9 Interrupts

RTS_interrupt

```
typedef enum {
    LEMMA_Synch_request,
    LEMMA_RTS_Msg_arrived,
    LEMMA_RTS_Clear_to_send,
    LEMMA_Immutable_complete,
    LEMMA_Mutable_complete
} LEMMA_interrupt_reason;

void RTS_interrupt(LEMMA_interrupt_reason reason, int arg)
```

This function is called from LEMMA to signal to the RTS as a result of detecting certain events during calls to LEMMA functions, and specifically in calls to `LEMMA_handle_IO`. `LEMMA_handle_IO` is intended to be called as a result of a SIGIO or SIGPOLL and therefore can be called at any time. The LEMMA cannot assume that the RTS is in a safe state, for example it may have allocated an object but not initialised it, so it cannot start a garbage collection.

`RTS_interrupt` should check the arguments and set a flag for events which are of interest. It should later test the flag when it is in a safe state. Apart from `LEMMA_Synch_request` the events are advisory and the RTS can choose to ignore them. The meaning of the events and the value of `arg` in each case, are given below.

`LEMMA_Synch_request`

An action has occurred which LEMMA was unable to process at that time. The RTS must call `LEMMA_synchronise` as soon as possible once it is in a safe state (i.e. it has initialised any objects which it has allocated). This occurs, for example, if another machine has requested a garbage collection. The `arg` argument is undefined.

`LEMMA_RTS_Msg_arrived`

An RTS message has arrived. The RTS should call `LEMMA_get_rts_message` to retrieve the message. The server number which sent the message is given in `arg`.

`LEMMA_RTS_Clear_to_send`

Buffer space is now available to send an RTS message to the server whose server number is given in `arg`. If the RTS has any messages which it wishes to send to that machine and which previously failed because of insufficient buffer space it should call `LEMMA_send_rts_message` to try to send them.

`LEMMA_Immutable_complete`

An asynchronous call to LEMMA_ensure_object_is_accessible has completed. The `arg` argument is undefined.

`LEMMA_Mutable_complete`

An asynchronous call to `LEMMA_load_mutable` or `LEMMA_assign_mutable` has succeeded. The token for the operation is given in `arg`. The RTS must still call `LEMMA_check_mutable_operation` to complete the operation and retrieve any result.

10 Awaiting External Activity

LEMMA_select

```
LEMMA_result LEMMA_select(fd_set *readfds, fd_set
     *writefds, fd_set *exceptfds, struct timeval
     *timeout);
```

This function performs most of the functions of the `select` system call. See the manual page for `select`. It returns if there is any activity on the file descriptors or the time expires. It may, however, mask off the SIGIO interrupt, so calls to this function should always include all the file descriptors on which activity is expected. This function also returns if there is any LEMMA activity which might affect the RTS. The RTS should check to see if there have been any interrupts.

`LEMMA_select` should be called if there is no activity and the Unix process would otherwise block.

11 Catastrophic Failure

RTS_crash

```
void RTS_crash(char *error_message)
```

Called when a catastrophic error occurs, usually when an internal consistency check has failed. The RTS should print out the error message, do any necessary clean-up and stop. The RTS should not rely on LEMMA working properly after this function has been called.