Journal of Universal Computer Science, vol. 16, no. 2 (2010), 221-245
submitted: 19/10/09, accepted: 10/12/09, appeared: 28/1/10 © J.UCS

An Extensible Interpreter Framework for Software Transactional Memory

Charlotte Herzeel, Pascal Costanza and Theo D'Hondt
(Software Languages Lab, Vrije Universiteit Brussel, Belgium
{charlotte.herzeel – pascal.costanza – tjdhondt}@vub.ac.be)

Abstract: Software transactional memory (STM) is a new approach for coordinating concurrent threads, for which many different implementation strategies are currently being researched. In this paper we show that if a language implementation provides *reflective access to explicit memory locations*, it becomes straightforward to both (a) build an STM framework for this language and (b) to implement STM algorithms using this framework. A proof-of-concept implementation in the form of a Scheme interpreter (written in Common Lisp) is presented.
Key Words: Software transactional memory, Lisp, memory location objects
Category: D.1.3, D.3.3

1 Introduction

Software transactional memory (STM) [Shavit and Touitou 95] is a novel approach for coordinating concurrent threads. It proposes the use of a transactional model for coordinating reads and writes of shared data in a multithreaded system. Without such a mechanism, the (relative) order of these reads and writes is undefined, during the execution of which a program can cause problems if two threads try to write the same memory location. Such problems are known as *data races* and are traditionally dealt with by the programmer by using low-level mechanisms such as *locks* for controlling the progress of threads. Programming with locks is known to be difficult because the programmer can easily write code that introduces mistakes such as *deadlocks* or code that does not easily compose. STM alleviates many of these problems by offering a well-defined protocol for managing reads and writes of shared data automatically.

An efficient implementation of STM is however *hard*, and numerous strategies have been proposed, but there is no definitive winner [Larus and Rajwar 07]. For example, an STM's *transaction granularity* determines the unit of storage over which the system operates (object or word/pointer-based). Other design decisions include the use of pessimistic or optimistic concurrency control, early or late conflict detection, direct or deferred memory updates, and so on. For a detailed taxonomy we refer to Larus and Rajwar's work [Larus and Rajwar 07]. Each of these options results in an STM that performs better for different applications.

A number of *benchmark suites* have been developed for assessing the different variations of STM algorithms [Cao Minh et al. 08, Kulkarni et al. 07]. Benchmark suites focus on getting comparable benchmark results, by providing sets of dedicated test applications that can be run without change for different STM algorithms. To make this work, a benchmark suite defines a so-called "generic" STM interface that is used in those test applications, so that the implementations of the STM algorithm can silently vary underneath. However, such benchmark suites typically don't provide reusable building blocks for implementing the actual STM algorithms, but leave the programmers of such algorithms on their own. The latter is the focus of a *STM framework* that provides common STM functionality and hooks. In this paper we propose such an STM framework.

Herlihy et al. previously proposed a framework for STM [Herlihy et al. 06], but their approach differs greatly from ours. Their framework, in line with other STM implementations we know of [Herlihy et al. 03, Harris and Fraser 03, Larus and Rajwar 07, Ringenburg and Grossman 05], is built on top of an existing compiler that was not designed for supporting STM. In contrast, we start by designing a language architecture from scratch that exposes the hooks for supporting STM as a plugin. This simplifies both the implementation of the framework itself as well as the use of the framework for plugging in different STM algorithms.

The contributions of this paper are:

- an analysis of the hooks a language implementation needs to provide for implementing STMs as plugins, and our solution that proposes to provide reflective access to memory locations for this purpose;
- an interpreter framework with such explicit memory locations as a proof of concept, here implemented for Scheme, but transferable to other languages;
- an implementation of three example STM algorithms as extensions of this framework to validate our approach.

2 Concepts of Software Transactional Memory

2.1 Multiprocessing with shared memory

In multithreaded programs, the execution of threads is typically synchronized using *locks*, which is a mechanism for temporarily granting threads exclusive access to shared resources, for example shared memory locations. Though locks can be used to avoid data races, programming with locks is notoriously difficult and alternative synchronization strategies are still an important research topic [Harris and Fraser 03]. Recently, software transactional memory was proposed.

The idea behind software transactional memory (STM) is to use transactions for coordinating the execution of concurrent threads[Larus and Rajwar 07]. Software transactions inherit the *atomicity* and *isolation* properties from database transactions. *Atomicity* requires a transactional piece of code to execute completely or, in case of failure, to pretend to never have been executed at all (i.e. any side effects are undone). Isolation requires the result of executing a transaction not to influence the result of other concurrently executing transactions. A correct implementation of these properties assures that transactions do not lead to data races.

STM has been realized both as libraries [Herlihy et al. 03, Herlihy et al. 06] and language extensions [Harris and Fraser 03, Ringenburg and Grossman 05]. STM libraries offer programmers APIs for making transactions, while language support for STM typically consist of a keyword `atomic` for delimiting a block of code that needs to execute transactionally. For example, if Scheme had an `atomic` construct, then a thread-safe implementation of the `insert` operation for a double linked list could look like the code below. The underlying STM implementation assures the code inside `atomic` executes transactionally.

```
(define insert (node new-node)
  (atomic (set-previous new-node node)
          (set-next new-node (next node))
          (when (not (null-node-p (next node)))
            (set-previous (next node) new-node))
          (set-next node new-node)))
```

2.2 Structure of an STM implementation

An STM algorithm monitors the reads and writes of memory executed within transactions, and implements an algorithm for checking whether any of these accesses causes a data race. In case there is a data race, the STM makes sure the conflicting execution is undone by rolling back one of the transactions.

Larus and Rajwar divide STM implementations into two categories: Deferred-update and direct-update STMs [Larus and Rajwar 07]. They differ strongly in the general implementation strategy. Deferred-update STM systems are implemented following a nonblocking synchronization strategy. When transactions access a memory location, they acquire a copy of its content and proceed execution in terms of the copy. Only when a transaction commits, the STM system replaces the content of the accessed memory locations with such copies. In case the STM detects a data race, transactions are cheaply rolled back, since their side effects are not yet global and hence do not need to be undone.

Conversely, direct-update systems rely on a blocking synchronization strategy. Transactions can temporarily get exclusive access to a memory location and side effects are performed instantly. A more expensive rollback mechanism than for deferred-update systems is needed, as the STM system needs to store old content of memory locations to be able to restore them on a rollback. However, in case there are few data race conflicts, such systems can be very efficient.

We claim that if a language implementation provides an explicit representation for memory locations, the implementation of both kinds of STMs is much easier to realize than when this is not the case. Consequently, we also claim that explicit memory locations are a key ingredient for a framework in which to express different kinds of STM. In what follows we sketch a design of a Scheme interpreter with explicit memory locations, and discuss the implementation of a direct-update and two deferred-update STM algorithms on top of the memory location abstraction. Afterwards, we discuss the inherent complexity of an STM implementation on top of a language implementation without an explicit representation for memory locations.

3 STM for a Scheme implemented in CLOS

Our experiment consists of extending a Scheme interpreter written in the Common Lisp Object System (CLOS [DeMichiel and Gabriel 87]) with explicit memory locations. The interpreter implements a non-trivial subset of Scheme. Additionally, it supports parallel variants of familiar constructs like `parallel-do`, `parallel-let`, etc as found in QLisp [Gabriel and McCarthy 84]. It also implements the `atomic` construct for executing a piece of code transactionally [Harris and Fraser 03]. Our interpreter is written using LispWorks[1] and relies on its multiprocessing package for threading and locking functionality. Our implementation is primarily meant to illustrate our claims, but does not focus on efficiency. We will discuss efficiency concerns in Section 6.

3.1 Transactional execution

Our interpreter extends the prototypical Lisp interpreter with a clause for evaluating *atomic* expressions. The code for `eval-atomic` is listed below. Note that `mp:*current-process*` is part of the LispWorks API for getting hold of the current active thread.

For evaluating an atomic expression, we put the current active thread into a transactional state (see `push-transactional-mode`) and let it evaluate the expression. Afterwards, `commit` is called for finalizing the transaction and restoring the thread to a non-transactional state for the rest of the execution (see `pop-transactional-mode`). For this, we made it possible to add a transactional state to LispWorks threads, which is accessible through the methods `push-transactional-mode`, `pop-transactional-mode` and `peek-transactional-mode`. The transactional state of a thread itself is modeled as a stack of transaction objects, for supporting the evaluation of nested atomic expressions.

[1] For LispWorks ®, see http://www.lispworks.com/.

```
(defmethod eval-atomic (exp env cont)
  (let ((transaction (make-transaction exp env cont)))
    (push-transactional-mode mp:*current-process* transaction)
    (eval exp env #'commit)))

(defmethod commit (result)
  (funcall (cont (pop-transactional-mode mp:*current-process*)) result))
```

Transactions are modeled as objects that store a reference to their thread of execution, and the interpreter's state at the time the transaction is created. The latter is needed for rolling back a transaction:

```
(defmethod roll-back ()
  (let ((transaction (pop-transactional-mode mp:*current-process*)))
    (eval-atomic (atomic-block transaction) (env transaction) (cont transaction))))
```

The methods for commit and roll-back shown here provide the default implementations for these operations. They do not by themselves deviate from normal execution without transactions. However, by defining them as methods, we have established a protocol for transactional execution: Client code that extends our interpreter with STM can override these two methods to include the extra functionality required for committing a transaction, and rolling it back.

3.2 Memory locations as objects

Memory locations are modeled as instances of the class `memory-location`, which defines a slot for storing a memory location's content. The class can be extended to hold additional information necessary for implementing a particular STM.

A method `make-memory-location` is the constructor for making new memory location objects. It takes the content of the memory location as an argument. The methods `memory-location-value` and `(setf memory-location-value)` are used to respectively read and write the content of a memory location object. The methods `registered-read` and `registered-write` implement a read or write of a memory location that is registered by the STM. Both registered and non-registered accesses to memory locations are necessary because some internal memory accesses must *not* be registered to implement STM correctly.

The memory location and transaction abstractions we discussed make up a complete STM framework. A detailed overview of the different classes and methods is given in Appendix A.

4 Explicit memory allocation and access in Scheme

For implementing STM, it must be possible to advise all possible reads and writes of memory. For Scheme, this means it should be possible to extend reads and writes of *variables*, *cons cells* and *vectors*, that is, there are no other *primitive* means for allocating and accessing memory. Fig. 1 gives an overview of the constructs in Scheme for manipulating variables, cons cells and vectors. We next identify the methods in the interpreter that implement these operations and open them up for extension. An overview of these methods is shown in Fig. 2.

	Allocation	Reading	Writing
Variables	`(define` *x obj*`)`	*x*	`(set!` *x obj*`)`
Cons cells	`(cons` *obj obj*`)`	`(car` *cons-cell*`)` `(cdr` *cons-cell*`)`	`(set-car!` *cons-cell obj*`)` `(set-cdr!` *cons-cell obj*`)`
Vectors	`(vector` *size*`)`	`(vector-ref` *vector idx*`)`	`(vector-set!` *vector idx obj*`)`

Figure 1: Allocating and accessing memory in Scheme

	Allocation	Reading	Writing
Variables	`make-binding`	`binding`	`set-binding`
Cons cells	`make-cl-cons`	`cl-list-car` `cl-list-cdr`	`cl-list-set-car` `cl-list-set-cdr`
Vectors	`make-cl-vector`	`cl-vector-ref`	`cl-vector-set`

Figure 2: Methods implementing memory allocation and access

4.1 Variable allocation and access

Variable bindings are stored in an *environment* structure, a dictionary-like structure allocated on the Common Lisp heap that maps variable names onto values. Internally, the latter mappings are represented using a structure `binding`.

Creating a new variable/value binding (for interpreting a `define`) is handled by a method `add-binding`: It calls `make-binding` to create a new instance of the structure `binding` and stores it into the global environment. Updating a variable/value binding (for interpreting a `set!`) is done by a method `set-binding`. Finally, looking up a variable (for interpreting a variable reference) is done by a method `binding`.

In a next step, we can now override these methods to make the memory locations referenced by variables explicit, by inserting explicit memory location objects in the bindings. The code for creating and accessing a binding is appropriately changed:

```
(defmethod make-binding :around ((atom atom) value)
  (list atom (make-memory-location value) 'binding))

(defmethod memory-location-of-value (binding) (second binding))

(defmethod binding-value :around (binding)
  (memory-location-value (memory-location-of-value binding)))

(defmethod (setf binding-value) :around (value binding)
  (setf (memory-location-value (memory-location-of-value binding)) value))
```

Note that we use `around` methods to override rather than specialize the original method definitions (they do not invoke `call-next-method`). This may seem superfluous, as we could just omit the original definitions, but then we would "lose" the original implementation of variable bindings without explicit memory locations. The latter would break the layered design of our plugin architecture and make the code conceptually incomplete. A less ad-hoc implementation can be realized in a language which has explicit support for implementing these kinds of software layers, such as ContextL (see Section 6).

Next, we override the methods `binding` and `set-binding`, which respectively implement variable lookup and update, to work on the new bindings. Variable lookup and update are operations that need to be monitored by the STM algorithm, hence the use of `registered-read` and `registered-write`, the methods we previously defined for monitored accesses.

```
(defmethod binding :around ((atom atom) (environment environment))
  (let ((binding (binding-mapping atom environment)))
    (registered-read (memory-location-of-value binding))))

(defmethod set-binding :around ((environment environment) (atom atom) (handle handle))
  (let ((binding (binding-mapping atom environment)))
    (if binding (registered-write (memory-location-of-value binding) handle)
      (error "Cannot assign to an undefined variable"))
    binding))
```

4.2 Vector allocation and access

Vectors are implemented using Common Lisp arrays, allocated on the Common Lisp heap. We represent vectors using a wrapper class `cl-vector` whose instances hold references to such Common Lisp arrays. A method called `make-cl-vector` is responsible for creating new vectors (for interpreting a `make-vector`). Methods `cl-vector-ref` and `cl-vector-set` implement reading and updating vector entries (for interpreting a `vector-ref` and `vector-set!`).

To allow advising of vector allocation and access, we now make the memory locations vectors reference explicit. In the code listed below, we override the constructor for vector objects. As previously discussed, vectors are represented by a class `cl-vector` that wraps a Common Lisp array. Here, we initialize the entries of the array with a memory location object:

```
(defmethod make-cl-vector :around (nr &optional initial-content)
  (if initial-content
      (make-instance 'cl-vector :cl-array (make-array nr :initial-contents
                                            (mapcar #'make-memory-location initial-content)))
    (make-instance 'cl-vector :cl-array (let ((new-array (make-array nr)))
                                          (dotimes (i nr)
                                            (setf (aref new-array i) (make-memory-location)))
                                          new-array))))
```

Next, we override `cl-vector-ref` and `cl-vector-set` to operate on the explicit memory locations:

```
(defmethod cl-vector-ref :around ((cl-vector cl-vector) nr)
  (let ((memory-location (aref (cl-array cl-vector) nr)))
    (registered-read memory-location)))

(defmethod cl-vector-set :around ((cl-vector cl-vector) nr val)
  (let ((memory-location (aref (cl-array cl-vector) nr)))
    (registered-write memory-location val)))
```

4.2.1 Cons cells and other data structures

Cons cells are essentially vectors of fixed length two, so they are implemented in a similar fashion as `cl-vector`, namely by providing a Common Lisp class for wrapping Common Lisp cons cells and providing corresponding methods for the respective operations on pairs. Similarly to what we did for vectors and variables, memory locations are introduced into the cons cell implementation, so that cons cell accesses can be advised to implement STM. Other data structures, like classes, can be supported in a similar fashion as extensions of the interpreter, or can be built on top of vectors and closures as user code in Scheme itself.

5 Plugging in STM implementations

5.1 Implementing a direct-update STM

Our first example STM is based on 2-phase locking with optimistic reads (as for example used in BSTM [Harris et al. 06]). When a transaction reads a memory location, the transaction takes a note of this. For writing a memory location, a transaction needs to acquire an exclusive lock. On acquiring the lock, the transaction first records a copy of the memory location's content and only then updates its content with the new value. The lock is released when the transaction successfully finishes. The STM checks for data races at well-defined times. *Write-after-read* conflicts are checked on reading a memory location by verifying that no other transaction has a lock on it. Additionally, when a transaction finishes, the STM checks for *read-after-write* conflicts by checking that none of the memory locations read by the transaction were updated afterwards. When there are no conflicts, the transaction finishes (commits) and releases all of its locks. Conversely, when a conflict is detected, the transaction rolls back, undoes any of the writes it performed, releases its locks and restarts. *Write-after-write* data races are avoided as transactions have to acquire an exclusive lock for writing a memory location, and these locks are only released when a transaction commits.

5.1.1 Memory location and transaction extensions

From the description above we derive the following extensions to our interpreter. We extend memory locations with a *version* for making it possible to check on commit whether a read memory location was updated by comparing its current version with the version on read. A memory location's version consists of a counter and a reference to the transaction that performed the last write.

```
(defclass versioned-memory-location (memory-location)
  ((lock    :initform (mp:make-lock)           :accessor memory-location-lock)
   (version :initform (make-instance 'version) :accessor memory-location-version)))
```

The code listed above shows the implementation of the class named `versioned-memory-location` that extends `memory-location` with slots for holding a lock and a version. The slots are initialized with default values, respectively a new lock and version object. The function `mp:make-lock` for creating a lock comes from the LispWorks `MP` package. The accessor `memory-location-value` for accessing a memory location's content remains unchanged. The constructor `make-memory-location` is overridden to create an instance of the class `versioned-memory-location`:

```
(defmethod make-memory-location :around (&optional value)
  (make-instance 'versioned-memory-location :value value))
```

We also extend transactions with a read and a write set. The sets are modeled as property lists, mapping each accessed memory location object onto a version object (in case of the read set) or a copy of the memory location's previous content (in case of the write set). Acessors `get-read-set` and `get-write-set` are defined for accessing the read or write set of a transaction.

5.1.2 Advising access of memory locations

In this STM, `registered-read` and `registered-write` are implemented as follows. `registered-read` makes a copy of the memory location's current version and, together with the memory location, pushes it onto the transaction's read set. Subsequently, it calls `locked-by-other-thread-p` to check if another thread holds a lock on the memory location: If so, there is a potential *write-after-read* data race, and the transaction is rolled back. Otherwise, the memory location's content is returned.

```
(defmethod registered-read ((mem-loc versioned-memory-location))
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (when transaction
      (setf (get-read-set transaction)
            (list* mem-loc (duplicate (memory-location-version mem-loc))
                   (get-read-set transaction)))
      (when (locked-by-other-thread-p mem-loc)
        (roll-back)))
    (memory-location-value mem-loc)))
```

`registered-write` calls `obtain-lock` for getting an exclusive lock on the memory location. Subsequently, it pushes the memory location and its current content on the transaction's write set. The latter is a sufficient "copy" since we register all memory writes by default. The next two expressions are responsible for increasing the version number and updating the "process-that-did-the-last-write." Subsequently the memory location's content is replaced by the new value.

```
(defmethod registered-write ((mem-loc versioned-memory-location) value)
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (if transaction
        (obtain-lock mem-loc
                     (lambda ()
                       (setf (get-write-set transaction)
                             (list* mem-loc (memory-location-value mem-loc)
                                    (get-write-set transaction)))
                       (setf (process-that-did-last-write (memory-location-version mem-loc))
                             transaction)
                       (incf (version-nr (memory-location-version mem-loc)))
                       (setf (memory-location-value mem-loc) value)))
      (setf (memory-location-value mem-loc) value))))
```

We also show the code for `obtain-lock` below, which tries to acquire the lock (via a call to `mp:process-lock`), but if that fails – because another transaction has the lock and waiting to get it takes too long – the transaction rolls back. `roll-back` removes the current transaction from the current process, undoes the writes it performed, releases its locks, removes the recorded read and write sets, and finally, the transaction is restarted (`call-next-method`).

```
(defmethod obtain-lock ((mem-loc memory-location) cont)
  (let* ((lock (memory-location-lock mem-loc))
         (lock-is-mine (mp:process-lock lock :timeout 3)))
    (if lock-is-mine (funcall cont) (roll-back))))

(defmethod roll-back :around ()
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (undo-writes-from-process transaction)
    (release-locks transaction)
    (call-next-method)))
```

The code for committing a transaction is shown below. It calls `verify-reads` to check for data races. Recall that the read set of a transaction is modeled as a property list: For each pair in that list, consisting of a memory location object and its version at the time it was read, we check if the current version of the memory location object is different from the old. If so, there is a data race and the commit fails.

```
(defmethod commit :around (result)
  (if (verify-reads)
      (progn
       (release-locks)
       (call-next-method))
   (roll-back)))

(defmethod verify-reads ()
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (loop for (memory-location version-on-read) on (get-read-set transaction) by #'cddr
          when (version-changed-p memory-location version-on-read) return nil
          finally (return T))))
```

This concludes the implementation of the STM algorithm based on 2-phase locking. Note that the implementation is purely an extension of our memory location and transaction abstractions: No other parts of the interpreter need to be changed to plug in the STM.

5.2 Implementing a deferred-update STM

The second example we implement is the DSTM system by Herlihy et al. [Herlihy et al. 03]. It is a lock-free, deferred-update STM that implements a nonblocking synchronization strategy. In DSTM, a memory location does not store *one* content, but *two*. It also stores a reference to the transaction that did the last write of the memory location. Depending on the status of that transaction – "active," "aborted" or "committed" – one content takes the role of the memory location's content before the last write (the "old" content) and the other plays the role of its current content (the "new" content).

When a transaction reads a memory location, DSTM checks the status of the transaction that performed the last write. If its status is "aborted," then the memory location's "old" content is returned. Otherwise, if the status is "committed," then the "new" value is returned. In both cases, the read is successful and recorded in the transaction's read set. Finally, when the status is "active," the memory location is in use by another transaction, and a conflict resolution is started to see if that transaction can keep using it or hands it over.

Similarly, for writing a memory location, DSTM checks the status of the transaction that performed the last write. Again, when the status is "active," the transactions negotiate for "ownership" of the memory location. Otherwise, when the status of the transaction that performed the last write is "committed" (or "aborted"), a new memory location object is created, with as "old" content a copy of the old memory location's "new" (or "old") content and as "new" content the write value. Additionally, the "transaction that did the last update" of the new memory location object is set to the transaction performing the write. Then, using a *compare-and-swap* operation, the old memory location object is atomically *replaced* by the newly created one.[2]

There are two places where DSTM checks for data races. On reading a memory location, DSTM checks for *write-after-read* data races by checking that the status of the transaction that did the last write is not "active." Otherwise, that transaction wrote a value the other transaction still had to read. On committing a transaction, DSTM checks for *read-after-write* data races by going through the transaction's read set and checking if any of the read memory locations was updated by another transaction in between. If so, the transaction aborts, setting its status to "abort." However, it is not necessary to undo any of the writes the aborted transaction performed, as the written memory locations still have a copy of the "old" content, and because the transaction's status is set to "abort," future accesses will return this "old" content. *Write-after-write* data races are avoided by DSTM, since a transaction can only obtain write access when no

[2] *compare-and-swap* is a known hardware primitive that atomically compares the content of a memory location to a value, and if they are the same, changes the content of the memory location to a new, given value.

other transaction is actively using the memory location, and the write access is only given up when the transaction's status changes to "commit" or "abort."

5.2.1 Memory location and transaction extensions

Given the above description, we need to extend memory location objects and transactions as follows. We create a new class `dstm-memory-location` for representing DSTM memory locations. We do not add new slots, but the idea is that a memory location's content is an instance of the class `content-unit`, as shown in the code below (see the initialization of the slot `memory-location-value` in `make-memory-location`). The class `content-unit` is a container for holding the version, the "old" content and the "new" content of a memory location. For convenience, we define methods `version`, `memory-location-new-content` and `memory-location-old-content` for accessing the the latter three objects directly from a memory location object (not shown).

```
(defmethod make-memory-location :around (&optional value)
  (make-instance
   'dstm-memory-location
   :memory-location-value (make-instance 'content-unit :new-value value)))

(defclass content-unit ()
  ((version :initarg :version :initform (make-instance 'version) :accessor version)
   (new-content :initarg :new-content :initform nil :accessor new-content)
   (old-content :initarg :old-content :initform nil :accessor old-content)))
```

The `memory-location-value` reader is overridden as shown below. It dispatches on the status of the transaction that performed the last write: If the status is "committed," the transaction returns the "new" content. In case it is "aborted," it returns the "old content." Otherwise, when the transaction that performed the last write is still active, the current transaction negotiates with that transaction to obtain access to the memory location. In our current implementation, the negotiation strategy is to simply wait, but any other strategy can be implemented here. Also note that in case there is no transaction that performed the last write – when a variable was only initialized, but never written – then `memory-location-value` also returns the "new" content of the memory location, which is set when defining a variable.

```
(defmethod memory-location-value ((mem-loc dstm-memory-location))
  (let ((last-writer (process-that-did-last-write (version mem-loc))))
    (cond ((or (null last-writer) (committed-p last-writer))
           (memory-location-new-content mem-loc))
          ((aborted-p last-writer)
           (memory-location-old-content mem-loc))
          (t (negotiate-for mem-loc last-writer)))))
```

We extend transactions with a read set and a status flag. A transaction can be in three states: When a transaction starts, its status is set to "active," on roll back it is set to "aborted" and on commit to "committed." The predicates `aborted-p`, `committed-p` and `active-p` are defined for querying the status of a transaction. Additionally, methods are defined for switching the status of a transaction (`change-status-to-aborted`, `change-status-to-committed` and `change-status-to-active`).

5.2.2 Advising access of memory locations

For DSTM, `registered-read` and `registered-write` are implemented as follows. The code for `registered-read` is quite straightforward: It makes an entry in the current transaction's read set, checks for (read after write) data races and if that succeeds, it returns the memory location's content, otherwise the transaction is rolled back:

```
(defmethod registered-read ((mem-loc dstm-memory-location))
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (when transaction
      (setf (get-read-set transaction)
            (list* mem-loc (duplicate (version mem-loc))
                   (get-read-set transaction)))
      (if (verify-reads) (memory-location-value mem-loc)
        (roll-back)))
    (memory-location-value mem-loc)))
```

The implementation of `registered-write` is a bit more tricky due the use of the `mp:compare-and-swap` operation.[3] We first get hold of the memory location's content unit (see `content-unit-before-write`) and we create a new content unit by calling `make-new-content-unit-from`. Subsequently, we try to replace the memory location's content with `new-content-unit` through the call to `mp:compare-and-swap`. For this, the latter fetches again what is in the slot `memory-location-value` of the memory location object and compares it to the previously fetched `content-unit-before-write`. When the *compare-and-swap* fails, because these two do not point to the same object anymore, we know that in between time, the memory location's content was updated by another transaction. To resolve this, the current transaction is rolled back and restarted.

```
(defmethod registered-write ((mem-loc dstm-memory-location) new-val)
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (if transaction
        (let ((content-unit-before-write (slot-value mem-loc 'memory-location-value))
              (new-content-unit
               (make-new-content-unit-from content-unit-before-write new-val transaction)))
          (unless (mp:compare-and-swap (slot-value mem-loc 'memory-location-value)
                                       content-unit-before-write new-content-unit)
          (roll-back)))
      (setf (memory-location-value mem-loc) new-val))))
```

On committing a transaction, we verify if the transaction is involved in a *read-after-write* conflict: See the call to `verify-reads` in the code below. If there is no conflict, the transaction's status is changed to "committed", otherwise the the transaction is rolled back.

[3] `mp:compare-and-swap` is introduced in LispWorks 6.0, which is not yet publicly available, but accessible to us for testing. An alternative implementation is to use a lock.

```
(defmethod commit :around (result)
  (if (verify-reads)
      (let ((transaction (peek-transactional-mode mp:*current-process*)))
        (change-status-to-committed transaction)
        (call-next-method))
    (roll-back)))
```

Rolling back a transaction in DSTM is quite cheap: We just change its status to "aborted." Then the transaction can safely restart. There is no need to roll back any of the side effects it performed, since memory locations store a copy of the content before the transaction performed any update, and that copy will from then on be accessed by other transactions.

```
(defun roll-back :around ()
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (change-status-to-aborted transaction)
    (call-next-method)))
```

That's it for the implementation of the DSTM algorithm. We stress again that the implementation is purely an extension of our memory location and transaction abstractions, and that there is no other parts of the interpreter that needs to be changed to plug in the STM.

5.3 Implementing a multiversion STM

The last example STM we discuss is a deferred-update STM based on multiversion concurrency control [Bernstein and Goodman 83], as for example used in the Clojure programming language [Volkmann 09]. In a multiversion STM, each memory location keeps track of its content history. For reading a memory location, a transaction looks up what the content was at the time the transaction was started. This way transactions that need to read a memory location that was already updated by a newer transaction, can still proceed correctly.

A memory location's access history consists of two parts: a *read history* and a *write history*. The *read history* is a list of transactions that read the memory location. The *write history* stores for every write on the memory location a pair that maps the writer transaction onto the new content. For reading a memory location, a transaction can query the write history for the content at the time the transaction was started. Note that a transaction does not directly operate on the access history of a memory location, rather it logs its accesses in a local history, which on commit is merged with the memory location's access history. Because of this, rolling back a transaction is a cheap operation.

Data races are either avoided, or handled at commit time. *Write-after-read* data races are avoided: A transaction that needs to read a memory location that was already written by a newer transaction, just looks up the old content in the memory location's write history. *Write-after-write* data races are checked on committing a transaction. If the transaction is trying to write a memory location that was already written by a newer transaction, then the commit fails and the transaction is rolled back. Similarly, *read-after-write* data races are handled on commit. There are two cases. Firstly, a transaction that read a memory location that was afterwards written by an earlier transaction, is rolled back. Secondly, a transaction that writes a memory location which was already read by a newer transaction, is also rolled back. Since it then gets a new timestamp, it then has a chance to run to a successful commit, cf. definitions of `eval-atomic` and `roll-back` in Section 3.1.

One drawback of multiversion STM is the memory overhead associated with storing the history of a memory location's content. Any realistic multiversion STM needs to implement some kind of garbage collection for keeping the access histories small. On the other hand, according to [Bernstein and Goodman 83], there should be less transaction rollbacks than in other transaction algorithms.

5.3.1 Memory location and transaction extensions

From the above description of multiversion STM, we derive the following extensions to our interpreter. We extend memory locations with a *write* and *read history* for recording the memory location's accesses. The read history is just a list of transaction timestamps, ordered in descending order. The write history is a list of pairs that each map a timestamp onto a value.

```
(defclass multiversion-memory-location (memory-location)
  ((lock :initform (mp:make-lock) :accessor lock)
   (write-history :initform '() :accessor write-history)
   (read-history :initform '() :accessor read-history)))
```

The code above shows the implementation of `multiversion-memory-location`, a class that extends `memory-location` with slots for holding the write and read history. Also note the slot `lock`, which we add for implementing atomic access of a memory location object. We specialize the accessors to operate on the write history:

```
(defmethod memory-location-value ((mem-loc multiversion-memory-location))
  (cl:cdr (cl:first (write-history mem-loc))))

(defmethod (setf memory-location-value) (value (mem-loc multiversion-memory-location))
  (push (cons 0 value) (write-history mem-loc)))
```

The "current" content of a memory location is the latest entry in the write history, as returned by `memory-location-value`. To update the content of the memory location, we add an entry to the write history. Note that we choose "0" as a timestamp for this entry, but this is just a stub value, as `(setf memory-location-value)` is only called when it is not important if the write happens transactionally. In addition to specializing the memory location accessors, we must also override the constructor `make-memory-location` so that it creates an instance of the class `multiversion-memory-location`.

We additionally extend transactions with a timestamp, a list of read memory locations and a list of written memory locations. The code below shows the implementation of the new transaction class named `multiversion-transaction`.

```
(defclass multiversion-transaction (stm-transaction)
  ((timestamp :initform (generate-timestamp) :accessor timestamp)
   (locks :initform '() :accessor locks)
   (read-memory-loactions :initform '() :accessor read-memory-locations)
   (written-memory-locations :initform '() :accessor written-memory-locations)))
```

The timestamp is unique for each transaction and defines an ordering relation between transactions. It is basically implemented with a counter. The slot `read-memory-locations` holds a list where each entry maps a memory location object object onto the length of the memory location's write history at the time the entry is created. Similarly, `written-memory-locations` represents a list where each entry consists of a memory location, a value and the length of the memory location's write history at the time the entry is created. As we explain in the next section, the latter is used for checking on commit if a written memory location was updated by another transaction. Note the slot `locks`: Its purpose is to hold a list of memory location locks so that commit can be implemented as an atomic operation.

5.3.2 Advising access of memory locations

In our implementation of multiversion STM, the methods `register-read` and `registered-write` are implemented as follows. `registered-read` first registers the read access with the transaction (`push`). Then it tries to find the content to return. It first checks if the transaction already wrote the memory location by trying to look up its value in the transaction's list of written memory locations (`find-if`). If this is the case, then the locally stored value is returned. Otherwise, the correct content to read is looked up in the memory location's write history (`history-lookup`). The function `history-lookup` takes the memory location's write history and the transaction's timestamp as arguments. It returns the entry with the largest timstamp that is smaller than or equal to the given timestamp. Note that all of this code is wrapped in a `mp:with-lock` call, which grabs the lock of the memory location being read. This is necessary to protect against data races from concurrent accesses to the write history of a memory location (in `push` and `history-lookup`).

```
(defmethod registered-read ((mem-loc multiversion-memory-location))
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (if transaction
        (mp:with-lock ((lock mem-loc))
          (push (cons mem-loc (length (write-history mem-loc)))
                (read-memory-locations transaction))
          (let ((write-entry (find-if (lambda (entry) (eql (car entry) mem-loc))
                                      (written-memory-locations transaction))))
            (if write-entry
                (cadr write-entry)
              (cdr (history-lookup (write-history mem-loc) (timestamp transaction))))))
      (memory-location-value mem-loc))))
```

The code for `registered-write` is listed below. It just adds an entry to the transaction's list of written memory locations (`push`), and therefore also needs to lock the memory location.

```
(defmethod registered-write ((mem-loc multiversion-memory-location) value)
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (if transaction
        (mp:with-lock ((lock mem-loc))
          (push (list mem-loc value (cl:length (write-history mem-loc)))
                (written-memory-locations transaction)))
      (setf (memory-location-value mem-loc) value))))
```

The code for committing a transaction is listed below. It calls `verify-reads` and `verify-writes` which check if the transaction we are trying to commit causes a data race. If the transaction does not cause any data races, the transaction's writes are made global (`commit-updates`) and the transaction is finalized (`call-next-method`). `commit-updates` merges the transaction's list of read and written memory locations with the respective memory locations' access histories.

```
(defmethod commit :around (result)
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (cond ((and (verify-reads transaction) (verify-writes transaction))
           (commit-updates transaction)
           (release-locks transaction)
           (call-next-method))
          (t
           (release-locks transaction)
           (roll-back)))))
```

Note the calls to `release-locks`. Committing a transaction must happen atomically, and, as we next discuss, `verify-reads` and `verify-writes` take locks on the memory locations accessed by the transaction: `release-locks` is there to free those locks again.

The code for checking whether the writes of a transaction cause any data races is shown below. `verify-writes` goes through a transaction's list of written memory locations. For each memory location, it grabs its lock (`mp:process-lock`). Then it performs two checks. First, it checks whether the memory location was updated by a transaction that started later (`later-transaction-updated-p`). If this is the case, as we discussed in our introduction on multiversion STM, this may cause a *write-after-write* data race. In that case, `verify-writes` yields false and causes a rollback in `commit`. The second check in `verify-writes` verifies if any newer transaction read the memory location. This may cause a *read-after-write* data race and consequently `verify-writes` fails.

```
(defmethod verify-writes ((transaction multiversion-transaction))
  (loop for entry in (written-memory-locations transaction)
        do (mp:process-lock (lock (car entry)))
        (push (lock (car entry)) (locks transaction))
        when (later-transaction-updated-p (car entry) transaction (caddr entry))
        return nil
        when (later-transaction-read-p (car entry) transaction (caddr entry))
        return nil
        finally (return T)))
```

The code for `later-transaction-updated-p` works as follows. The method goes through the entries of the write history of the memory location to check if there is one with a timestamp that is smaller than the transaction's. It doesn't go through the entire write list. It only checks the write list up until the entries that were already in the memory location's write list when the transaction tried to write it. The code for `later-transaction-read-p` is very similar to `later-transaction-updated-p` and is omitted here for the sake of conciseness.

```
(defmethod later-transaction-updated-p ((mem-loc multiversion-memory-location)
                                        (transaction multiversion-transaction)
                                        nr-of-writes-at-access)
  (let ((bad-update nil))
    (do ((ctr (- (cl:length (write-history mem-loc)) 1) (- ctr 1))
         (updates-to-check (write-history mem-loc) (rest updates-to-check)))
        ((or (= ctr (- nr-of-writes-at-access 1))
             bad-update) bad-update)
      (setf bad-update (< (timestamp (first updates-to-check) (timestamp transaction)))))))
```

For completeness, the code for `verify-reads` is listed below. It has the same structure as `verify-writes`. Finally, the implementation for `roll-back` is unchanged from the default: It just restarts the current transaction. No writes need to be undone: Before commit, these are only visible to the transaction itself.

```
(defmethod verify-reads ((transaction multiversion-transaction))
  (loop for entry in (read-memory-locations transaction)
        do (mp:process-lock (lock (car entry)))
        (push (lock (car entry)) (locks transaction))
        when (earlier-transaction-updated-p (car entry) transaction (cdr entry))
        return nil
        finally (return T)))
```

This concludes our implementation of multiversion STM. Again, we stress that the implementation is purely an extension of our memory location and transaction abstractions. There are no other parts of the interpreter that need to be changed to plug in the STM. Also note that we added the multiversion STM later, after our original paper at ELS 2009, and that no changes were necessary to the memory location and transaction abstractions.

6 Discussion and Related Work

The implementation of the STMs we just discussed shows that it is indeed possible to use our memory location model for implementing STMs as plugins. We now address our original claim, that it is much harder to implement STM as part of a language that doesn't provide reflective access to explicit memory locations.

Assume we try to implement STM on top of plain Common Lisp. Common Lisp provides some predefined data structures, like variables, cons cells, vectors, and arrays, and ways of defining new user-defined data structures using defstruct, defclass and define-condition. This means that the number of potential datatypes in Common Lisp is open-ended, which is true for most general-purpose languages.

It is possible to implement STM algorithms for Common Lisp by deciding to support one or more specific kinds of datastructures, for example by using custom slot accessors in the CLOS MOP [Ton-That 07, Costanza et al. 09], or by shadowing accessors for cons cells. However, because of the open-endedness of Common Lisp, such STM libraries cannot provide complete coverage of all possible data structures.

This is also true for DSTM2, which is the only other framework with support for implementing STM algorithms we are aware of [Herlihy et al. 06]. Their approach is implemented as a library for Java that takes advantage of Java's reflection capabalities and its class loader architecture to create new classes at runtime for speciallly annotated Java interface definitions. These new classes contain pairs of getter and setter methods with additional behavior as required by the various STM algorithms, much like the adaptations of accesses to memory locations that we described above. New STM algorithms can be plugged in that provide templates for new such getter and setter methods. However, such STM algorithms can only operate on *instance variables* of Java classes, but not, for example, on class variables or array entries. This restriction is due to the fact that Java does not provide reflective access to its internal representation of memory locations.

Our interpreter framework provides a single abstraction for memory locations, and guarantees that all memory accesses always go through a handful of well-defined accessor methods. So it is sufficient to override these accessor methods once to plug in new STM algorithms, without having to do this for each and every kind of data structure over and over again.

A current drawback of our approach is that it doesn't pay a lot of attention to efficiency concerns: It introduces overhead because of the wrapping of internal representations of data structures and because each memory access goes through a generic function call. This is due to the fact that we focused on illustrating the essential idea of explicit memory locations as our primary first goal. It has been shown in the past that reflective architectures like the one presented in this paper can indeed be implemented efficiently [Kiczales et al. 91], but it remains to be shown to what extent we can do this for our approach as well.

As a first step, we have already taken the approach described in this paper and reimplemented it on top of ContextL [Costanza et al. 09]. ContextL is an extension of CLOS that provides support for expressing behavioral variations of programs in terms of layers that can be activated and deactivated with dynamic scope. The idea to implement the approach in ContextL is based on two observations: One is that the interpreter framework presented in this paper is already a layered design, consisting of the interpreter of the core language in one layer, the introduction of explicit memory locations in a second layer, and one or more STM algorithms in a third layer. Each of these layers extends the functionality of the respective layer below by extending well-defined hooks in the architecture of the overall framework. The extensions can be understood as concerns that cut across various operationally distinct events in a program, i.e., read and write accesses, entering and leaving atomic blocks, and commits and rollbacks. Therefore, ContextL's layers provide a good mechanism to modularize these crosscutting concerns.

The second observation is that the behavior of slot accesses depends indeed on context: Slot accesses inside transactions behave differently from slot accesses outside of transactions. In the interpreter framework described in this paper, this distinction is described by way of `if` statements, but in [Costanza et al. 09], we show how layer activation and deactivation at transaction boundaries can be used instead to switch between these different slot access semantics. Furthermore, the distinction between *registered* and *non-registered* slot accesses can also be expressed in terms of behavioral variations that can be selected by way of ContextL's layer activation and deactivation.

The integration of our STM framework in ContextL has the already described disadvantage of being able to treat only CLOS/ContextL slots transactionally, but not other kinds of memory locations. On the other hand, CLOS can be implemented efficiently [Kiczales et al. 91], and ContextL uses efficient implementation techniques for context-dependent method dispatch and layer activations [Costanza and Hirschfeld 07]. We have already implemented the Delaunay Mesh Refinement algorithm from the Lonestar test suite [Kulkarni et al. 09] in this ContextL-based realization of our approach, and this benchmark already confirms previous reports about the improved performance of direct-update over deferred-update algorithms for STM [Herlihy et al. 03], among other things. See [Costanza et al. 09] for more details. We plan to use this implementation as a basis for further investigations of the relative performance of our approach.

7 Conclusions and Future Work

In this paper we have shown that if a language implementation provides reflective access to explicit memory locations, it becomes straightforward to implement both (a) a framework for software transactional memory, and (b) different STM algorithms using this framework. We have presented a proof-of-concept implementation in the form of a Scheme interpreter with such explicit memory locations and subsequently implemented two deferred-update and one direct-update STM algorithm in terms of the memory location abstraction to back our claims. The fact that we expressed three very different STM algorithms in our framework, confirms that our approach is stable enough for a wide range of STM algorithms.

For future work we intend to investigate efficient implementation techniques, by removing overhead that is caused (a) by unnecessary wrappers in the internal representation of basic data types and (b) by unnecessary generic function calls for accessing memory locations that are never accessed by more than one thread. We also plan to implement more standard benchmarks for STMs [Cao Minh et al. 08, Kulkarni et al. 09]. Even without a more efficient implementation, we can already gain interesting insights from them, by counting the number of unnecessary rollbacks under different STM algorithms and under different, simulated access patterns in competing threads.

Acknowledgements

We thank Dave Fox, Usha Millar and Martin Simmons from LispWorks ® for letting us use an alpha version of LispWorks 6.0, and for their support on the `MP` library. We also thank Richard Gabriel and Guy Steele for their comments on earlier drafts of this paper. Charlotte Herzeel's research is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). This work is partially funded by the Research Foundation - Flanders (FWO).

References

[Bernstein and Goodman 83] Philip A. Bernstein and Nathan Goodman, Multiversion concurrency control—theory and algorithms, *ACM Transactactions on Database Systems*, 1983.

[Cao Minh et al. 08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis and Kunle Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing *IISWC '08, Proceedings*, 2008

[Costanza et al. 09] Pascal Costanza, Charlotte Herzeel, Theo D'Hondt: Context-oriented Software Transactional Memory in Common Lisp, *Proceedings of the Dynamic Languages Symposium 2009*, ACM, 2009.

[Costanza and Hirschfeld 07] Pascal Costanza and Robert Hirschfeld: Reflective layer activation in ContextL. *Proceedings of the 2007 ACM Symposium on Applied Computing*, ACM, 2007.

[DeMichiel and Gabriel 87] Linda DeMichiel and Richard P. Gabriel, The Common Lisp Object System: An Overview, *ECOOP'87: Proceedings of the European Conference on Object-oriented Programming*, 1987.

[Gabriel and McCarthy 84] Richard P. Gabriel and John McCarthy, Queue-based multi-processing LISP, *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, 1984.

[Harris and Fraser 03] Tim Harris and Keir Fraser, Language Support for Lightweight Transactions, OOPSLA'03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems and Languages, 2003.

[Harris et al. 06] Tim Harris, Mark Plesko, Avraham Shinnar and David Tarditi, Optimizing Memory Transactions, *PLDI'06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, 2006.

[Herlihy et al. 06] Maurice Herlihy, Victor Luchangco and Mark Moir,A Flexible Framework for Implementing Software Transactional Memory, *OOPSLA 2006, Proceedings*, 2006

[Herlihy et al. 03] Maurice Herlihy, Victor Luchangco, Mark Moir and William N. Scherer, III, Software Transactional Memory for Dynamic-sized Data Structures, *PODC '03: Proceedings*, 2003

[Kiczales et al. 91] Gregor Kiczales, Jim des Rivieres and Daniel Bobrow, The Art of the Metaobject Protocol, *MIT Press, Cambridge, MA, USA*, 1991.

[Kulkarni et al. 09] Milind Kulkarni, Martin Burtscher, Calin Cascaval and Keshav Pingali, Lonestar: A suite of parallel irregular programs, *IEEE International Symposium on Performance Analysis of Systems and Software*, IEEE, 2009.

[Kulkarni et al. 07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew, Optimistic parallelism requires abstractions, *PLDI '07, Proceedings*, 2007

[Larus and Rajwar 07] James R. Larus and Ravi Rajwar, Transactional Memory, *Morgan Claypool Publishers, USA*, 2007.

[Ringenburg and Grossman 05] Michael F. Ringenburg and Dan Grossman, AtomCaml: First-class Atomicity via Rollback, *ICFP'05: Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming*, 2005.

[Shavit and Touitou 95] Nir Shavit and Dan Touitou, Software Transactional Memory, *PODC '95, Proceedings*, 1995

[Ton-That 07] Hoan Ton-That, *http://common-lisp.net/project/cl-stm/*, January 2007.

[Volkmann 09] R. Mark Volkmann, Software Transactional Memory, *http://java.ociweb.com/mark/stm/article.html*, April 2009.

A Overview of the STM framework

commit *Generic Function*

Syntax:

commit *result*

Arguments and Values:

The *result* argument can be any Lisp value implementing a Scheme object; it is the result for an atomic expression.

Purpose:

This method is called to finalize execution of an atomic expression. By default, it just removes the transactional state from the currently active process (by calling `pop-transactional-mode`) and continues computation. This method needs to be overridden to plug in an STM algorithm, for example, to release the locks held by a transaction.

make-memory-location *Generic Function*
Syntax:
make-memory-location &*optional value*
Arguments and Values:
The method returns a new **memory-location** object where the optional argument *value* specifies the initial content.
Purpose:
This method is a constructor for making memory location objects. By default, it makes an instance of the class **memory-location** for which it fills in the **content** slot. The method must be specialized for subclasses of **memory-location**.

---

make-transaction *Generic Function*
Syntax:
make-transaction *exp env cont*
Arguments and Values:
The argument *exp* is the atomic expression whose evaluation triggers transactional execution. The argument *env* is the environment in which the atomic expression is evaluated. The argument *cont* is the continuation of the atomic expression, i.e. the computation that comes after evaluation of the atomic expression. It returns a **transaction** object that saves this state.
Purpose:
This method is a constructor for making transaction objects. By default, it makes an instance of the class **transaction** and fills in the **exp**, **env** and **cont** slots. The method must be specialized for subclasses of **transaction**.

---

memory-location *Class*
Slots:
This class has one slot **content**. Accessors are **registered-read**, **registered-write**, **memory-location-value** and **(setf memory-location-value)**.
Purpose:
This class is there to represent memory locations as explict objects. It is subclassed by any STM algorithm that needs to store additional information about memory locations beside their content.

---

memory-location-value *Generic Function*
Syntax:
memory-location-value *memory-location*
Arguments and Values:
The method returns the content of a memory location object.
Purpose:
This method is a getter method for reading the content of a memory location object. This method is called whenever it is important that the access is not registered by the STM algorithm. This method can be specialized for subclasses of **memory-location**.

(setf memory-location-value) *Generic Function*
Syntax:
(setf make-memory-location) *value memory-location*
Arguments and Values:
The method sets the content slot of a give *memory-location* object to *value*. *memory-location* is an instance of **memory-location**; *value* can be any Lisp value implementing a Scheme object.
Purpose:
This method is a setter for the content of a memory location object. This method is used whenever it is important that the access is *not* registered by the STM algorithm. Specialize this method for subclasses of **memory-location**.

---

peek-transactional-mode *Generic Function*
Syntax:
peek-transactional-mode *process*
Arguments and Values:
The *process* argument is a LispWorks process object. Typically, this will be **mp:*current-process***. This method returns the top of the *process'* transaction stack.
Purpose:
This method is called to get hold of the transactional state that is currently active for a process. It does not need to be specialized to implement an STM algorithm.

---

pop-transactional-mode *Generic Function*
Syntax:
pop-transactional-mode *process*
Arguments and Values:
The *process* argument is a LispWorks process object. Usually, this will be **mp:*current-process***. This method removes and returns the top of the *process'* transaction stack.
Purpose:
This method is called to get hold of the transactional state that is currently active for a process. It also removes that transactional state from the process. It does *not* need need to be specialized to implement an STM algorithm.

---

push-transactional-mode *Generic Function*
Syntax:
push-transactional-mode *process transaction*
Arguments and Values:
The *process* argument is a LispWorks process object. The *transaction* argument is an instance of class **transaction**.
This method puts *transaction* on the top of the *process'* transaction stack.
Purpose:
This method is called when the interpreter is evaluating an atomic expression. It puts a transaction object onto the process' transaction stack. It does *not* need need to be specialized to implement an STM algorithm.

registered-write *Generic Function*
Syntax:
registered-write *memory-location value*
Arguments and Values:
The method updates the content slot of a **memory-location** object with *value*, which can be any Lisp value that implements a Scheme object.
Purpose:
This method is a setter method for updating the content of the memory location object. This method is called for accesses that are registered by the STM algorithm, and can be specialized for subclasses of **memory-location**.

---

registered-read *Generic Function*
Syntax:
registered-read *memory-location*
Arguments and Values:
memory-location is an instance of class **memory-location**.
Purpose:
This method is a getter method for reading the content of the memory location object. This method is called for accesses that are registered by the STM algorithm, and can be specialized for subclasses of **memory-location**.

---

roll-back *Generic Function*
Syntax:
roll-back
Arguments and Values:
n/a
Purpose:
This method is called to restart a transaction. By default, it removes the transactional state from the currently active process (by calling **pop-transactional-mode**) and restarts evaluation with the atomic expression, environment and continuation state stored in the discarded transaction. This method needs specialization for specific STM implementations, for example to undo any changes made by a transaction.

---

transaction *Class*
Slots:
This class has a slot **thread**, for storing a reference to the LispWorks process that is executing transactionally. There is also a slot **exp** for storing the atomic expression that triggers transactional evaluation, and a slot **env** and **cont** for storing the environment and continuation for evaluating the atomic expression.
Purpose:
This class is there to represent transactional state. Its slots store the state that is necessary to restart execution of an atomic expression. It must be subclassed to implement STM-specific transactions.