A Domain Specific Language for Composable Memory Transactions in Java

André Rauber Du Bois[1] and Marcos Echevarria[1]

PPGInf - Programa de Pós-Graduação em Informática,
Universidade Católica de Pelotas
CEP: 96010-000, Pelotas-RS, Brazil
{dubois, marcosge}@ucpel.tche.br

Abstract. In this paper we present CMTJava, a domain specific language for *composable memory transactions* [7] in Java. CMTJava provides the abstraction of *transactional objects*. Transactional objects have their fields accessed only by special get and set methods that are automatically generated by the compiler. These methods return *transactional actions* as a result. A transactional action is an action that, when executed, will produce the desired effect. Transactional actions can only be executed by the `atomic` method. Transactional actions are first class values in Java and they are composable: transactions can be combined to generate new transactions. The Java type system guarantees that the fields of transactional objects will never be accessed outside a transaction. CMTJava supports the `retry` and `orElse` constructs from STM Haskell. To validate our design we implemented a simple transactional system following the description of the original Haskell system. CMTJava is implemented as a state passing monad using BBGA closures, a Java extension that supports closures in Java.

1 Introduction

Multi-core machines are now available everywhere. The traditional way to program these machines using threads, shared memory and locks for synchronization is difficult and has many pitfalls [16, 22, 24]. A promising approach to program multi-core machines is to use *software transactional memory* (STM). In this approach, accesses to shared memory are performed inside transactions, and a transaction is guaranteed to be executed atomically with respect to other concurrently executed transactions. Transactions can be implemented using *optimistic concurrency*: all transactions are executed concurrently with no synchronization, all writes and reads to shared memory are performed in a *transaction log*. When a transaction finishes, it validates its log to check if it has seen a consistent view of memory, and if so, its changes are committed to memory. If validation fails the log is discarded and the transaction is executed again with a fresh log. With atomic transactions the programmer still has to worry when critical sections should begin and end. But although having to delimit critical sections, the programmer does not need to worry about a locking protocol, that can induce problems like deadlocks.

In this paper, we present CMTJava, a domain specific language to write *composable memory transactions* in Java. CMTJava brings the idea of composable memory transactions in the functional language Haskell [7] to a object oriented context.

The main characteristics of the CMTJava system are:

- CMTJava provides the abstraction of *Transactional Objects*. Transactional objects have their fields accessed only by special get and set methods that are automatically generated by the compiler. These methods return *transactional actions* as a result. A transactional action is an action that, when executed, will produce the desired effect. Java's type system guarantees that the fields of transactional objects can only be accessed inside transactions. Transactions can only executed with the `atomic` method. The `atomic` method takes a transaction as an argument and executes it atomically with respect to all other memory transactions. As transactions can not be executed outside a call to `atomic`, properties like *atomicity* (the effects of a transaction must be visible to all threads all at once) and *isolation* (during the execution of a transaction, it can not be affected by other transactions) are always manteined.
- Transactional actions are first class values in Java, they can be passed as arguments to methods and can be returned as the result of a method call. Transactions are composable: they can be combined to generate new transactions. Transactions are composed using `STM` blocks (Section 2), and we also support the `retry` construct, that allows possibly-blocking transactions to be composed sequentially and the `orElse` construct that allows transactions to be composed as alternatives.
- Transactions are implemented as a state passing monad (Section 3.3). Most of the simplicity of the implementation comes from the fact that we use a Java extension for closures to implement CMTJava. Hence, this paper is also a case support for closures in Java 7. We describe translation rules that translate CMTJava to pure Java + closures. Although the system was implemented in Java, the ideas could also be applied to any object oriented language that supports closures, e.g., C#.

The paper is organized as follows. First, in Section 2, CMTJava is presented through simple examples. Section 3, describes the implementation of the language primitives using a *state passing* monad. A simple prototype implementation of transactions is also described. Section 4 discusses related work and finally Section 5 presents conclusions and directions for future work.

2 Examples

In this section we present simple examples with the objective of describing how to program with CMTJava.

2.1 The Dinning Philosophers

In this section, an implementation of the *dinning philosophers problem* is given using the CMTJava DSL. The first class to be defined is `Fork`:

```
class Fork implements TObject{
   private volatile Boolean fork = true;}
```

The `Fork` class has only one field and Java's `volatile` keyword is used to guarantee that threads will automatically see the most up-to-date value in the field. The `TObject` interface works as a *hint* to the compiler, so it will generate automatically the code needed to access this class in transactions. It also generates two methods to access each field of the class. For the `Fork` class it will generate the following methods:

```
STM<Void> setFork(Boolean fork);
STM<Boolean> getFork();
```

The `setFork` and `getFork` methods are the only way to access the `fork` field. A value of type `STM<A>` represents a *transactional action* that when executed will produce a value of type `A`.

We can now define the `Philosopher` class as in Figure 1. The `STM{...}`

```
class Philosopher implements Runnable{

 Fork right;
 Fork left;
 int id;

 Philosopher(int i, Fork r, Fork l)
 {
   id = i;
   right = r;
   left = l;
 }

 public void run()
 {
   STM<Void> t1 = STM{
            acquireFork(r);
            acquireFork(l)
            }
   atomic(t1);

   System.out.println("Philosopher " + id +
                      " is eating!!!");
   STM<Void> t2 = STM{
            releaseFork(r);
            releaseFork(l)
            }
   atomic(t2);
 }
 (...)
}
```

Fig. 1. The `Philosopher` class

block works like the `do` notation in STM Haskell [7]: it is used to compose transactions. The notation `STM{` a_1`;...;` a_n`}` constructs a STM action by glueing together smaller actions a_1`;...;`a_n in sequence. Using `STM` blocks it is possible to implement the `STM<Void> acquireFork(Fork f)` and `STM<Void> releaseFork(Fork f)` methods that are missing on Figure 1:

```
STM<Void> acquireFork(Fork f)
{
  STM<Void> r = STM{
     Boolean available <- f.getFork();
     if (!available)
        retry()
     else
        f.setFork(false)
     }
  return r;
}
```

The `acquireFork` method first checks the current state of the `Fork` using the `getFork()` method. If the fork is not available it blocks the transaction by calling `retry`. If the fork was available, it is set to `false`, i.e., not available. The call to `retry` will *block* the transaction, i.e., the transaction will be aborted and restarted from the beginning. The transaction will not be re-executed until at least one of the fields of the `TObjects` that it has accessed is written by another thread. In the case of the `acquireFork` method, when it calls `retry` the philosopher will be suspended until a neighbor philosopher puts the fork back in the table. The variables created inside a transactional action are *single assignment* variables. These variables are used only to carry the intermediate state of the transaction being constructed and do not need to be logged by the runtime system supporting transactions. To emphasize that these variables are different than the ordinary Java variable a different symbol for assignment is used (`<-`).

The `releaseFork` method has a simple implementation:

```
STM<Void> releaseFork(Fork f)
{
  return f.setFork(true);
}
```

It returns a transaction that when executed will set the fork field to `true`. When a philosopher calls `releaseFork` it already holds the fork so there is no need to check its current state.

2.2 The `orElse` method

CMTJava also provides the `orElse` method that allows the composition of transactions as *alternatives*

```
public static <A> STM<A> orElse (STM<A> t1, STM<A> t2)
```

`orElse` takes two transactions as arguments and returns a new transaction. The transaction

```
orElse(t1,t2);
```

will first execute `t1`; if it retries then `t1` is discarded and `t2` is executed. If `t2` also retries then the whole transaction will retry.

In Figure 2 a simple implementation of a class representing a bank account is given. Using `orElse` we could extend the `Account` class with the following method:

```
public static STM<Void>withdraw2Accounts(Account c1, Account c2)
{
  return orElse(c1.withdraw(1000.0), c2.withdraw(1000.0));
}
```

The `withdraw2Accounts` method first tries to withdraw `1000.0` from `c1`. If there is not enough money in that account it tries to withdraw the same amount from `c2`

```
class Account implements TObject{
 private volatile Double Balance;

 public STM<Void> withdraw(Double n)
 {
   STM<Void> t = STM{
           Double b <- getBalance();
           if (b < n)
             retry()
           else
             setBalance(b-n)
   };
   return t;
 }

 public STM<Void> deposit (Double n)
 {
   return STM{
   Double b <- readBalance();
   writeBalance(b + n)
   };
 }

 public static STM<Void> transfer(Account a1, Account a2, Double money)
 {
   return STM{
     a1.withdraw(money);
     a2.deposit(money)
   }
 }
}
```

Fig. 2. The `Account` class

3 Implementation

3.1 Java Closures

To implement CMTJava we used *BGGA Closures*, a Java extension that supports *anonymous functions* and *closures* [2]. This extension is a prototype implementation of a JSR proposal to add closures to Java 7 [3].

In BBGA, an anonymous function can be defined using the following syntax:
{ *formal parameters* => *statements expression*}
where *formal parameters*, *statements* and *expression* are optional. For example, { `int x => x + 1` } is a function that takes an integer and returns its value incremented by one. An anonymous function can be invoked using its `invoke` method:

```
String s = {=> "Hello!"}.invoke();
```

An anonymous function can also be assigned to variables:

```
{int => void} func = {int x => System.out.println(x)};
```

The variable `func` has type {`int => void`}, i.e., a *function type* meaning that it can be assigned to a function from `int` to `void`. Function types can also be used as types of arguments in a method declaration.

A *closure* is a function that captures the bindings of free variables in its lexical context:

```
public static void main(String[] args) {
      int x = 1;
      {int=>int} func = {int y => x+y };
      x++;
      System.out.println(func.invoke(1)); // will print 3
    }
```

A closure can use variables of the enclosing scope even if this scope is not active at the time of closure invocation e.g., if a closure is passed as an argument to a method it will still use variables from the enclosing scope where it was created.

3.2 Monads

A monad is a way to structure computations in terms of values and sequences of computations using those values [1]. A monad is used to describe computations and how to combine these computations to generate new computations. For this reason monads are frequently used to embed domain specific languages in functional languages for many different purposes, e.g., I/O and concurrency [20], Parsers [13], controlling robots [19], and memory transactions [7]. A monad can be implemented as an abstract data type that represents a container for a computation. These computations can be created and composed using three basic operations: *bind*, *then* and *return*. For any monad `m`, these functions have the following type in Haskell:

```
bind :: m a -> (a -> m b )-> m b
then :: m a -> m b -> m b
return :: a -> m a
```

The type `m a` represents a computation in the monad `m` that when executed will produce a value of type `a`. The `bind` and `then` functions are used to combine computations in a monad. `bind` executes its first argument and passes the result to its second argument (a function) to produce a new computation. `then` takes two computations as arguments and produces a computation that will execute them one after the other. The `return` function creates a new computation from a simple value.

The next section presents the implementation of these three operations for the `STM` monad in Java.

3.3 The STM Monad

The monad for STM actions is implemented as a *state passing monad*. The state passing monad is used for threading a state through computations, where each computation returns an altered copy of this state. In the case of transactions, this state is the meta-data for the transaction, i.e., its log.

The `STM` class is implemented as follows:

```
class STM<A> {
    { Log => STMResult<A> } stm;

    STM ({ Log => STMResult<A> } stm) {
      this.stm = stm;
    }
}
```

The `STM` class is used to describe transactions, and it has only one field: a function representing the transaction. A transaction `STM<A>` is a function that takes a `Log` as an argument and returns a `STMResult<A>`. The `Log` represents the current state of the transaction being executed and `STMResult` describes the new state of the transaction after its execution.

The `STMResult<A>` class has three fields:

```
class STMResult<A> {

    A result;
    Log newLog;
    int state;

   (...)
}
```

the first field (`result`) is the result of executing the `STM` action, the second (`newLog`) is the resulting log, and the third is the current `state` of the transaction. Executing a `STM` action can put the transaction in one of two states: it is either `VALID` meaning the the transaction can continue, or the state can be set to `RETRY` meaning that the `retry` construct was called and the transaction must be aborted.

The `bind` method is used to compose transactional actions:

```
public static <A,B> STM<B> bind ( STM<A> t, {A => STM<B> } f ) {
  return new STM<B> ( {Log l1 =>
         STMResult<A> r = t.stm.invoke(l1);
         STMResult<B> re;
         if (r.state == STMRTS.VALID) {
           STM<B> nst = f.invoke(r.result);
           re = nst.stm.invoke(r.newLog);
          } else {
           re = new STMResult(null, r.newLog, STMRTS.RETRY);
          }
         re
     } );
}
```

The `bind` method takes as arguments an `STM<A>` action `t` and a function `f` of type {`A => STM<B>` } and returns as a result a new `STM<B>` action. The objective of `bind` is to combine `STM` actions generating new actions. The resulting `STM` action takes a log (`l1`) as an argument and invokes the `t` action by passing the log to it (`t.stm.invoke(l1)`). If the invocation of `t` did not retry (i.e., its state is `VALID`) then `f` is called generating the resulting `STM<B>`. Otherwise the execution flow is abandoned and `bind` returns a `STMResult` with the state set to `RETRY`.

The `then` method is implemented in a very similar away

```
public static <A,B> STM<B> then ( STM<A> a, STM<B> b ) {

    return new STM<B> ( {Log l1 =>
           STMResult<A> r = a.stm.invoke(l1);
           STMResult<B> re;
           if (r.state == STMRTS.VALID) {
              re = b.stm.invoke(r.newLog);
           } else {
              re = new STMResult(null, r.newLog, STMRTS.RETRY);
           }
           re
      } );
}
```

The `then` method is a sequencing combinator: it receives as arguments two `STM` actions and returns an action that will execute them one after the order.

Finally, the `stmreturn` method is used to *insert* any object `A` into the `STM` monad:

```
public static <A> STM<A> stmreturn (A a) {
  return new STM<A>({ Log l => new STMResult(a,l,STMRTS.VALID) });
}
```

The `stmreturn` method is like Java's `return` for `STM` blocks. It takes an object as an argument and creates a simple transaction that returns this object as a result. It can also be used to create new objects inside a transaction. For example, the `insert` method returns a transaction that inserts a new element at the end of a linked list:

```
class Node implements TObject{
  private volatile Integer value;
  private volatile Node next;
  (...)
  public STM<Void> insert(Integer v)
  {
   return STM {
     Node cnext <- getNext();
     if( cnext == null) {
       STM{
         Node r <- stmreturn (new Node(v,null));
         setNext(r)
        }
     }else
        cnext.insert(v)
     }
  }
}
```

3.4 `STM` blocks

`STM` blocks are translated into calls to `bind` and `then` using the translation rules given in Figure 3. Translation rules for `STM` blocks are very similar to the translation rules for the `do` notation described in the Haskell report [21].

```
STM{ type var <- e; s} = STMRTS.bind( e, { type var => STM { s }})

STM{ e ; s }           = STMRTS.then( e,  STM{ S })

STM{ e }               = e
```

Fig. 3. Basic translation schemes for STM blocks

For example, the following implementation of a `deposit` method:

```
public STM<Void> deposit (Account a, Double n)
 {
   return STM{
              Double balance <- a.getBalance();
              a.setBalance(balance + n)
            };
 }
```

is translated to

```
public STM<Void> deposit (Account a, Double n)
 {
   return STMRTS.bind(a.getBalance(), { Double balance =>
          a.setBalance(balance + n)});
 }
```

3.5 Compiling TObjects

A class like `Fork` given in section 2.1 is compiled to the class in Figure 4, using the translation rules given in Appendix A.

```
class Fork implements TObject {

   private volatile Boolean fork = true;
   private volatile PBox<Boolean> forkBox =
       new PBox<Boolean> ( {Boolean b => fork = b; } , { => fork} );

       public STM<Void> setFork (Boolean b) {
            return new STM<Void>({Log l =>
            LogEntry<Boolean> le = l.contains(forkBox);
            if(le!=null) {
              le.setNewValue(b);
            } else {
              l.addEntry(new LogEntry<Boolean>(forkBox,fork,b));
            }
            new STMResult(new Void(), l,STMRTS.VALID)} );
       }

       public STM<Boolean> getFork() {
            return new STM<Boolean> ({Log l =>
            Boolean result;
            LogEntry<Boolean> le = l.contains(forkBox);
            if(le!=null) {
              result = le.getNewValue();
            } else {
              result =fork;
              l.addEntry(new LogEntry<Boolean>(forkBox,fork,fork));
            }
            new STMResult(result, l,STMRTS.VALID)} );
       }
}
```

Fig. 4. The Fork class after compilation

Every `TObject` has a `PBox` for each of its fields. A `PBox` works like a pointer to the field. The constructor of the `PBox` class takes two arguments: one is a function that updates the current value of a field and the other is a function that returns the current value of a field. This functions are used by `atomic` to commit a transaction (Section 3.8).

The `setFork` and `getFork` methods are used to access the `fork` field in a transaction. The `setFork` method first verifies if the `PBox` for the field is already present in the transaction log using the `contains` method of the `Log`. A log is a collection of log entries:

```
class LogEntry<A> {
    PBox<A> box;
    A oldValue;
    A newValue;
(...)
}
```

Each `LogEntry` represents the current state of a field during a transaction. It contains a reference to the field's `PBox`, the original content of the field (`oldValue`) and the current value of the field (`newValue`). If the log already contains an entry for `fork` then this entry is updated so its `newValue` field contains a reference to the argument of the set method. If the log contains no entry for `fork` a new entry is added where the `oldValue` contains the current content of `fork` and `newValue` is set to the argument of `setFork`.

`getFork` will first check if there is an entry for `fork` in the log. If so, it returns the current value of the `newValue` field of the entry, otherwise it adds a new entry to the log where both `oldValue` and `newValue` contain the current value in the `fork` field.

3.6 The `retry` method

The `retry` method has a simple implementation:

```
public static STM<Void> retry() {
   return new STM<Void>( { Log l =>
            new STMResult(new Void(),l,STMRTS.RETRY)});
}
```

It simply stops the current transaction by changing the state of the transaction to `STMRTS.RETRY`. When a transaction returns `STMRTS.RETRY` the transaction is aborted (see the implementations for `bind` and `then` in Section 3.3).

The `atomic` method is responsible for restarting aborted transactions (Section 3.8).

3.7 The `orElse` method

The `orElse` method is used to compose transactions as alternatives. To implement `orElse`, proper nested transactions are needed to isolate the execution of

the two alternatives. If the first alternative calls `retry`, all the updates that it did must be invisible while executing the second alternative.

The `orElse` method is implemented in a similar way to `orElse` in STM Haskell. Each alternative is executed using a new log called *nested log*. All writes to fields are recorded in the nested log while reads must consult the nested log and the log for the enclosing transaction. If any of the alternatives finishes without retrying, its nested log is merged with the transaction log and a `VALID` `STMResult` is returned containing the merged log. If the first alternative calls `retry`, then the second is executed using a new nested log. If the second alternative also calls `retry`, all three logs are merged and an `STMResult` containing the merged log and state set to `RETRY` is returned. All logs must be merged so that the aborted transaction will be executed again when one of the fields accessed in the transaction is modified by other transaction (Section 3.8). The reader should notice that, if both alternatives that retried modified the same filed of a `TObject`, the merged log will contain only one entry for that field. That does not matter as the merged log will be never used for committing, since the transaction was aborted. The final log is used to decide when the transaction should be restarted (Section 3.8).

3.8 The `atomic` method

An `STM` action can be executed by calling `atomic`:

```
public static <A> A atomic (STM<A> t)
```

The atomic method takes as an argument a transaction and executes it atomically with respect to other concurrent calls to atomic in other threads.

In order to execute a transactional action, the `atomic` method generates a fresh `Log` and then invokes the transaction:

```
Log l = STMRTS.generateNewLog();
STMResult<A> r = t.stm.invoke(l);
```

As described before, an invoked transaction will return a `STMResult` containing the result of executing the transaction, a log and the state of the transaction that can be either `VALID` or `RETRY`. A `Log` contains, besides log entries, a global lock that is used by `atomic` to update shared data structures when committing.

If the state of the executed transaction is `RETRY`, `atomic` acquires the global lock and validates the log of the transaction. To validate a log, each log entry is checked to see if its `oldValue` contains the same object that is currently in the field of the `TObject`. The current content of a field is accessed using the `PBox` in the log entry. If the log is invalid, the transaction is automatically restarted with a fresh log. If the log is valid, the transaction is only restarted once at least one of the fields that it accessed is modified by another thread. To do that, `atomic` first creates a `SyncObject` for the transaction. A `SyncObject` has two methods:

```
public void block();
public void unblock();
```

When a thread calls the `block` method of a `SyncObject`, the thread will block until some other thread calls the `unblock` method of the same object. If a thread calls `unblock`, and the blocked thread was already unblocked, the method simply returns with no effect. If a thread calls `unblock` before `block` was called, the thread waits for the call to `block` before continuing. Every `PBox` also contains a list of `SyncObjects` for the threads waiting for that field to be updated. After creating the `SyncObject`, it is added to the list of blocked threads in every `PBox` present in the transaction log. Then the transaction releases the global lock and calls the `block` method of the `SyncObject`, suspending the transaction. When the transaction is awaken, meaning that at least one of the `TObjects` in its log was update, it will be executed again with a fresh log.

If the state of the transaction executed by `atomic` is `VALID`, `atomic` acquires the global lock and validates the transaction log. If valid, the transaction commits by modifying all `TObjects` with the content of its log. It also awakes all threads waiting for the commit. This is done by calling the `unblock` method of the `SyncObjects` contained in the `PBox`es in the log. If the transaction log is invalid, the log is abandoned and the transaction is executed again with a fresh log.

4 Related Work

Transactional memory was initially studied as a hardware architecture [10, 23, 5]. Software transactional memory [25] is the idea of implementing all transactional semantics in software. Most works on STM in Java provide low level libraries to implement transactions [9, 8, 17]. Harris and Fraser [6] provide the first language with constructs for transactions. Their Java extension gives an efficient implementation of Hoare's conditional critical regions [11] through transactions, but transactions could not be easily composed. The Atomos language [4] is a Java extension that supports transactions through atomic blocks and also the `retry` construct to block transactions. Transactions are supported by the Transactional Coherence and Consistency hardware transactional memory model (TCC) [18], and programs are run on a simulator that implements the (TCC) architecture. No support composing transactions as alternatives is given.

CMTJava builds on work done in the Haskell language, by bringing the idea of composable memory transactions into an object oriented context. STM Haskell [7], is a new concurrency model for Haskell based on STM. Programmers define *transactional variables* (`TVar`s) that can be read and written using two primitives:

```
readTVar :: TVar a -> STM a
writeTVar :: Tvar a -> a -> STM a
```

The `readTVar` primitive takes a `TVar` as an argument and returns an STM action that, when executed, returns the current value of the `TVar`. The `writeTVar` primitive is used to write a new value into a `TVar`. In CMTJava, each field of a transactional object works as a `TVar` and each field has its get/set method that work just like `readTVar` and `writeTVar`. In STM Haskell, `STM` actions can be composed together using the same `do` notation used to compose `IO` actions in Haskell:

```
addTVar :: TVar Int -> Int -> STM ()
addTVar tvar i = do
                  v <- readTvar tvar
                  writeTVar tvar (v+i)
```

STM blocks in CMTJava are just an implementation of the `do` notation available in Haskell. The STM system described here is very similar to the one described in the STM Haskell paper, the main difference being that STM Haskell is implemented to run on uniprocessors. This simplifies the implementation as all calls to the C functions that support transactions in the runtime system are run without interruption. The simple implementation of CMTJava given in this paper can be executed on multi-core machines, as we use a global lock to avoid races during commits, at the cost of a huge bottleneck imposed by the lock.

In [12], an implementation of composable memory transactions using pure Haskell is given. CMTJava uses an implementation of the `STM` monad that is very similar to the one described in this paper, although CMTJava's implementation of the low level mechanisms to support transactions is very different, as it is based on the STM Haskell runtime system.

5 Conclusions and Future Work

In this paper we have presented CMTJava, a domain specific language for composable memory transactions in Java, in the style of STM Haskell. CMTJava guarantees that fields of transactional objects will only be accessed inside transactions. CMTJava is translated to pure Java + closures, that are supported by the BBGA Java extension [2]. Although the system was implemented in Java, the ideas could also be applied to any object oriented language that supports closures.

There are many directions for future work. A semantics for the language could be provided by extending Feather Weight Java [14] with the semantics for composable memory transactions in Haskell. Exceptions inside an `atomic` call could be handled in the same way as in STM Haskell.

The prototype implementation of the STM system presented in this paper is very naive and is given only to illustrate the concepts presented and as a proof of concept. Every time a field of a `TObject` is accessed the system must search in an array of `LogEntries` for the right entry before using the content of the field. The single lock used for committing transactions is also a bottleneck in the system. We plan to re-implement CMTJava in the future using a low level library for transactions in Java (e.g., [8]) or using other optimized technique for the implementation of transactional memory (a survey on the subject is given in [15]).

The source code for CMTJava can be obtained by contacting the authors.

Acknowledgment The authors would like to thank CNPq/Brazil for the financial support provided.

References

1. All About Monads . WWW page, http://www.haskell.org/all_about_monads/html/-index.html, December 2008.
2. Java Closures. WWW page, http://www.javac.info/, December 2008.
3. JSR Proposal: Closures for Java . WWW page, http://www.javac.info/consensus-closures-jsr.html, December 2008.
4. B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The ATOMOS transactional programming language. *ACM SIGPLAN Notices*, 41(6):1–13, June 2006.
5. L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. volume 32, page 102, New York, NY, USA, 2004. ACM.
6. T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, Nov. 2003.
7. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*. ACM Press, 2005.
8. Herlihy, Luchangco, and Moir. A flexible framework for implementing software transactional memory. *SPNOTICES: ACM SIGPLAN Notices*, 41, 2006.
9. Herlihy, Luchangco, Moir, and Scherer. Software transactional memory for dynamic-sized data structures. In *PODC: 22th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2003.
10. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
11. C. A. R. Hoare. Towards a theory of parallel programming. *Operating System Techniques*, pages 61–71, 1972.
12. F. Huch and F. Kupke. A high-level implementation of composable memory transactions in concurrent haskell. In *IFL*, pages 124–141, 2005.
13. G. Hutton and E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.
14. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–459, May 2001.
15. J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
16. E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
17. V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. revised, University of Rochester, Computer Science Department, May 2006.
18. A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *Proc. 14th International Conference on Parallel Architecture and Compilation Techniques (14th PACT'05)*, pages 63–74, Saint Louis, MO, USA, Sept. 2005. IEEE Computer Society.
19. J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 91–105, London, UK, 1998. Springer-Verlag.
20. S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.

21. S. Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 1(13), 2003.
22. S. Peyton Jones. *Beautiful Concurrency*. O'Reilly, 2007.
23. R. Rajwar, M. Herlihy, and K. K. Lai. Virtualizing transactional memory. In *ISCA*, pages 494–505. IEEE Computer Society, 2005.
24. R. Rajwar and G. James. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125, 2003.
25. Shavit and Touitou. Software transactional memory. *DISTCOMP: Distributed Computing*, 10, 1997.

Appendix A: Translation Rules for TObjects

Translation rules for `TObjects` are given as Haskell functions. These functions take as arguments Strings (i.e., names of variables and their types) and return as result a String representing the code to be generated. Strings in Haskell are represented using double quotes and the (`++`) operation is used to concatenate two strings.

PBox objects are generated using the `genPBox` function:

```
type Type = String
type VarName = String

genPBox :: Type -> VarName -> VarName -> String
genPBox vtype var parName =
        "private volatile PBox<" ++ vtype ++ "> " ++
         var ++ "PBox = new PBox<" ++ vtype ++"> ({" ++
         vtype ++ " " ++ parName ++ "  => " ++
         var ++ " = " ++ parName ++" ; } , { => "++var++"});"
```

The parameters for `genPBox` are the type of the field of the transactional object (`vtype`), the name of the field `var`, and a random name for the parameters of the closures being defined `parName`.

Set methods for `TObjects` are generated using `genSetMethod`:

```
genSetMethod :: Type -> VarName -> VarName -> VarName -> String
genSetMethod vtype var parName boxName =
 "public STM<Void> set"++var++"( "++vtype++" "++parName++")\n"++
 "{\n" ++
 " return new STM<Void>({Log l => \n" ++
 " LogEntry<"++vtype++"> le = l.contains("++boxName++");\n"++
 " if(le!=null) {\n" ++
 "   le.setNewValue("++parName++");\n" ++
 " } else {\n" ++
 "   l.addEntry(new LogEntry<"++vtype++">("++boxName++","++
     var++","++parName++"));\n"++
 " }\n" ++
 " new STMResult(new Void(), l,STMRTS.VALID)} );\n" ++
 "}\n"
```

Get methods for `TObjects` are generated using `genGetMethod`:

```
genGetMethod :: Type -> VarName -> VarName -> VarName -> String
genGetMethod vtype var parName boxName =
 "public STM<"++vtype ++"> get"++var++"() {\n"++
 " return new STM<"++vtype++"> ({Log l =>\n"++
 "    "++vtype++ " result;\n"++
 "    LogEntry<"++vtype++"> le = l.contains("++boxName++");\n"++
 "          if(le!=null) {\n"++
 "            result = le.getNewValue();\n"++
 "          } else {\n"++
 "            result ="++var++";\n"++
 "            l.addEntry(new LogEntry<"++vtype++">("++boxName++
                    ","++var++","++var++"));\n"++
 "          }\n"++
 "          new STMResult(result,l,STMRTS.VALID)} );\n"++
 "}\n"
```