

Figure 8.4: The hierarchy of atomic actions

8.2.3 Programming with atomic actions

Starting with the next section, we give the programming techniques for shared-state concurrency using atomic actions. We introduce the concepts gradually, starting with locks. We refine locks into monitors and transactions. Figure 8.4 shows the hierarchical relationships between these three concepts.

- Locks allow to group little atomic operations together into big atomic operations. With a reentrant lock, the same lock can guard discontinuous parts of the program. A thread that is inside one part can reenter the lock at any part without suspending.
- Monitors refine locks with wait points. A *wait point* is a pair of an exit and a corresponding entry with no code in between. (Wait points are sometimes called *delay points* [6].) Threads can park themselves at a wait point, just outside the lock. Exiting threads can wake up parked threads.
- Transactions refine locks to have two possible exits: a normal one (called *commit*) and an exceptional one (called *abort*). The exceptional exit can be taken at any time during the transaction. When it is taken, the transaction leaves the execution state unchanged, i.e., as it was upon entry.

Figure 8.5 summarizes the principal differences between the three concepts. There are many variations of these concepts that are designed to solve specific problems. This section only gives a brief introduction to the basic ideas.

Reasoning with atomic actions

Consider a program that uses atomic actions throughout. Proving that the program is correct consists of two parts: proving that each atomic action is correct (when considered by itself) and proving that the program uses them correctly. The first step is to show that each atomic action, e.g., lock, monitor, or transaction, is correct. Each atomic action defines an ADT. The ADT should have an

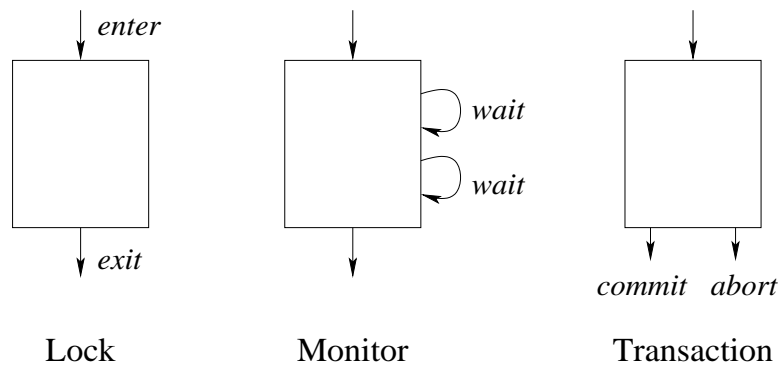


Figure 8.5: Differences between atomic actions

invariant assertion, i.e., an assertion that is true when there is no thread inside the ADT. This is similar to reasoning with stateful programs and active objects, except that the ADT can be accessed concurrently. Because only one thread can be inside the atomic action at a time, we can still use mathematical induction to show that the assertion is an invariant. We have to prove two things:

- When the ADT is first defined, the assertion is satisfied.
- Whenever a thread exits from the ADT, the assertion is satisfied.

The existence of the invariant shows that the atomic action is correct. The next step is to show that the program using the atomic actions is correct.

8.2.4 Further reading

There are many good books on concurrent programming. The following four are particularly well-suited as companions to this book. They give more practical techniques and theoretical background for the two concurrent paradigms of message-passing and shared-state concurrency. At the time of writing, we know of no books that deal with the third concurrent paradigm of declarative concurrency.

Concurrent Programming in Java

The first book deals with shared-state concurrency: *Concurrent Programming in Java, Second Edition*, by Doug Lea [111]. This book presents a rich set of practical programming techniques that are particularly well-suited to Java, a popular concurrent object-oriented language (see Chapters 7 and 8). However, they can be used in many other languages including the shared-state concurrent model of this book. The book is targeted towards the shared-state approach; message passing is mentioned only in passing.

The major difference between the Java book and this chapter is that the Java book assumes threads are expensive. This is true for current Java implementations. Because of this, the Java book adds a conceptual level between threads and

procedures, called *tasks*, and advises the programmer to schedule multiple tasks on one thread. If threads are lightweight this conceptual level is not needed. The range of practical programming techniques is broadened and simpler solutions are often possible. For example, having lightweight threads makes it easier to use active objects, which often simplifies program structure.²

Concurrent Programming in Erlang

The second book deals with message-passing concurrency: *Concurrent Programming in Erlang*, by Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding [9]. This book is complementary to the book by Doug Lea. It presents a rich set of practical programming techniques, all based on the Erlang language. The book is entirely based on the message-passing approach.

Concurrent Programming: Principles and Practice

The third book is *Concurrent Programming: Principles and Practice*, by Gregory Andrews [6]. This book is more rigorous than the previous two. It explains both shared state and message passing. It gives a good introduction to formal reasoning with these concepts, using invariant assertions. The formalism is presented at just the right level of detail so that it is both precise and usable by programmers. The book also surveys the historical evolution of these concepts and includes some interesting intermediate steps that are no longer used.

Transaction Processing: Concepts and Techniques

The final book is *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter [64]. This book is a successful blend of theoretical insight and hard-nosed practical information. It gives insight into various kinds of transaction processing, how they are used, and how they are implemented in practice. It gives a modicum of theory, carefully selected to be relevant to the practical information.

8.3 Locks

It often happens that threads wish to access a shared resource, but that the resource can only be used by one thread at a time. To help manage this situation, we introduce a language concept called *lock*, to help control access to the resource. A lock dynamically controls access to part of the program, called a *critical region*. The basic operation of the lock is to ensure exclusive access to the critical region, i.e., that only one thread at a time can be executing inside it. If the shared resource is only accessed from within the critical region, then the lock can be used to control access to the resource.

²Special cases of active objects are possible if threads are expensive, see e.g., Section 5.5.1.

```

fun {NewQueue}
  X in
    q(0 X X)
end

fun {Insert q(N S E) X}
  E1 in
    E=X|E1 q(N+1 S E1)
end

fun {Delete q(N S E) X}
  S1 in
    S=X|S1 q(N-1 S1 E)
end

```

Figure 8.6: Queue (declarative version)

The shared resource can be either inside the program (e.g., an object) or outside it (e.g., an operating system resource). Locks can help in both cases. If the resource is inside the program, then the programmer can guarantee that it cannot be referenced outside the critical region, using lexical scoping. This kind of guarantee can in general not be given for resources outside of the program. For those resources, locks are an aid to the programmer, but he must follow the discipline of only referencing the resource inside the critical region.

There are many different kinds of locks that provide different kinds of access control. Most of them can be implemented in Oz using language entities we have already seen (i.e., cells, threads, and dataflow variables). However, a particularly useful kind of lock, the thread-reentrant lock, is directly supported by the language. The following operations are provided:

- {NewLock L} returns a new lock.
- {IsLock X} returns **true** if and only if X references a lock.
- **lock X then** ⟨S⟩ **end** guards ⟨S⟩ with lock X. If no thread is currently executing any statement guarded by lock X, then any thread can enter. If a thread is currently executing a guarded statement, then the same thread can enter again, if it encounters the same lock in a nested execution. A thread suspends if it attempts to enter a guarded statement while there is another thread in a statement guarded by the same lock.

Note that **lock X then** ... **end** can be called many times with the same lock X. That is, the critical section does not have to be contiguous. The lock will ensure that at most one thread is inside any of the parts that it guards.

```

fun {NewQueue}
  X C={NewCell q(0 X X)}
  proc {Insert X}
    N S E1 in
      q(N S X|E1)=@C
      C:=q(N+1 S E1)
  end
  fun {Delete}
    N S1 E X in
      q(N X|S1 E)=@C
      C:=q(N-1 S1 E)
      X
  end
in
  queue(insert:Insert delete:Delete)
end

```

Figure 8.7: Queue (sequential stateful version)

8.3.1 Building stateful concurrent ADTs

Now that we have introduced locks, we are ready to program stateful concurrent ADTs. Let us approach this in steps. We give a systematic way to transform a declarative ADT to become a stateful concurrent ADT. We also show how to modify a sequential stateful ADT to become concurrent.

We illustrate the different techniques by means of a simple example, a queue. This is not a limitation since these techniques work for any ADT. We start from a declarative implementation and show how to convert this to a stateful implementation that can be used in a concurrent setting:

- Figure 8.6 is essentially the declarative queue of Section 3.4.4. (For brevity we leave out the function `IsEmpty`.) Delete operations never block: if the queue is empty when an element is deleted, then a dataflow variable is returned which will be bound to the next inserted element. The size `N` is positive if there are more inserts than deletes and negative otherwise. All functions have the form `Qout={QueueOp Qin ...}`, taking an input queue `Qin` and returning an output queue `Qout`. This queue will work correctly in a concurrent setting, insofar as it can be used there. The problem is that the order of the queue operations is explicitly determined by the program. Doing these queue operations in different threads will *ipso facto* cause the threads to synchronize. This is almost surely an undesired behavior.
- Figure 8.7 shows the same queue, but in a stateful version that encapsulates the queue's data. This version cannot be used in a concurrent setting without some changes. The problem is that encapsulating the state requires to *read* the state (`@`), do the operation, and then to *write* the new state (`:=`).

```

fun {NewQueue}
  X C={NewCell q(0 X X)}
  L={NewLock}
  proc {Insert X}
    N S E1 in
      lock L then
        q(N S X|E1)=@C
        C:=q(N+1 S E1)
      end
    end
  fun {Delete}
    N S1 E X in
      lock L then
        q(N X|S1 E)=@C
        C:=q(N-1 S1 E)
      end
      X
    end
in
  queue(insert:Insert delete:Delete)
end

```

Figure 8.8: Queue (concurrent stateful version with lock)

If two threads each do an insert, then both reads may be done before both writes, which is incorrect. A correct concurrent version requires the read-operation-write sequence to be atomic.

- Figure 8.8 shows a concurrent version of the stateful queue, using a lock to ensure atomicity of the read-operation-write sequence. Doing queue operations in different threads will not impose any synchronization between the threads. This property is a consequence of using state.
- Figure 8.9 shows the same version, written with object-oriented syntax. The cell is replaced by the attribute `queue` and the lock is implicitly defined by the `locking` property.
- Figure 8.10 shows another concurrent version, using an exchange to ensure atomicity. Since there is only a single state operation (the exchange), no locks are needed. This version is made possible because of the single-assignment property of dataflow variables. An important detail: the arithmetic operations $N-1$ and $N+1$ must be done *after* the exchange (why?).

We discuss the advantages and disadvantages of these solutions:

- The declarative version of Figure 8.6 is the simplest, but it cannot be used as a shared resource between independent threads.

```

class Queue
  attr queue
  prop locking

  meth init
    queue:=q(0 X X)
  end

  meth insert(X)
    lock N S E1 in
      q(N S X|E1)=@queue
      queue:=q(N+1 S E1)
    end
  end

  meth delete(X)
    lock N S1 E in
      q(N X|S1 E)=@queue
      queue:=q(N-1 S1 E)
    end
  end
end

```

Figure 8.9: Queue (concurrent object-oriented version with lock)

- Both concurrent versions of Figure 8.8 and 8.10 are reasonable. Figure 8.8’s use of a lock is more general, since a lock can be used to make atomic *any* set of operations. This version can be written with an object-oriented syntax, as shown in Figure 8.9. Figure 8.10’s version with exchange is compact but less general; it is only possible for operations that manipulate a single data sequence.

8.3.2 Tuple spaces (“Linda”)

Tuple spaces are a popular abstraction for concurrent programming. The first tuple space abstraction, called *Linda*, was introduced by David Gelernter in 1985 [59, 30, 31]. This abstraction plays two very different roles. From a theoretical viewpoint, it is one of the first models of concurrent programming. From a practical viewpoint, it is a useful abstraction for concurrent programs. As such, it can be added to any language, thus giving a concurrent version of that language (e.g., C with Linda is called C-Linda). A tuple space abstraction is sometimes called a *coordination model* and a programming language that contains a tuple space abstraction is sometimes called a *coordination language*. In its basic form, the abstraction is simple to define. It consists of a multiset \mathcal{TS} of tuples with three basic operations:

```

fun {NewQueue}
  X C={NewCell q(0 X X)}
  proc {Insert X}
    N S E1 N1 in
      {Exchange C q(N S X|E1) q(N1 S E1)}
      N1=N+1
  end
  fun {Delete}
    N S1 E N1 X in
      {Exchange C q(N X|S1 E) q(N1 S1 E)}
      N1=N-1
      X
  end
in
  queue(insert:Insert delete:Delete)
end

```

Figure 8.10: Queue (concurrent stateful version with exchange)

- {TS write(T)} adds the tuple T to the tuple space.
- {TS read(L T)} waits until the tuple space contains at least one tuple with label L. It then removes one such tuple and binds it to T.
- {TS readnonblock(L T B)} does not wait, but immediately returns. It returns with B=**false** if the tuple space contains no tuple with label L. Otherwise, it returns with B=**true**, removes one tuple with label L and binds it to T.

This slightly simplifies the usual formulation of Linda, in which the read operation can do pattern matching. This abstraction has two important properties. The first property is that it provides a *content-addressable memory*: tuples are identified only by their labels. The second property is that the readers are *decoupled* from the writers. The abstraction does no communication between readers and writers other than that defined above.

Example execution

We first create a new tuple space:

```
TS={New TupleSpace init}
```

In TS we can read and write any tuples in any order. The final result is always the same: the reads see the writes in the order they are written. Doing {TS write(foo(1 2 3))} adds a tuple with label foo and three arguments. The following code waits until a tuple with label foo exists, and when it does, it removes and displays it:

```
thread {Browse {TS read(foo $)}} end
```



```

fun {NewQueue}
  X TS={New TupleSpace init}
  proc {Insert X}
    N S E1 in
      {TS read(q q(N S X|E1))}
      {TS write(q(N+1 S E1))}
  end
  fun {Delete}
    N S1 E X in
      {TS read(q q(N X|S1 E))}
      {TS write(q(N-1 S1 E))}
      X
  end
in
  {TS write(q(0 X X))}
  queue(insert:Insert delete:Delete)
end

```

Figure 8.11: Queue (concurrent version with tuple space)

The following code immediately checks if a tuple with label `foo` exists:

```
local T B in {TS readnonblock(foo T B)} {Browse T#B} end
```

This does not block, so it does not need to be put in its own thread.

Implementing a concurrent queue

We can show yet another implementation of a concurrent queue, using tuple spaces instead of cells. Figure 8.11 shows how it is done. The tuple space `TS` contains a single tuple `q(N S E)` that represents the state of the queue. The tuple space is initialized with the tuple `q(0 X X)` that represents an empty queue. No locking is needed because the `read` operation atomically removes the tuple from the tuple space. This means that the tuple can be considered as a unique token, which is passed between the tuple space and the queue operations. If there are two concurrent `Insert` operations, only one will get the tuple and the other will wait. This is another example of the token passing technique introduced in Section 8.2.2.

Implementing tuple spaces

A tuple space can be implemented with a lock, a dictionary, and a concurrent queue. Figure 8.12 shows a simple implementation in object-oriented style. This implementation is completely dynamic; at any moment it can read and write tuples with any labels. The tuples are stored in a dictionary. The key is the tuple's label and the entry is a queue of tuples with that label. The capitalized methods `EnsurePresent` and `Cleanup` are private to the `TupleSpace` class and

```

class TupleSpace
  prop locking
  attr tupledict

  meth init tupledict:={NewDictionary} end

  meth EnsurePresent(L)
    if {Not {Dictionary.member @tupledict L}}
    then @tupledict.L:={NewQueue} end
  end

  meth Cleanup(L)
    if {@tupledict.L.size}==0
    then {Dictionary.remove @tupledict L} end
  end

  meth write(Tuple)
    lock L={Label Tuple} in
      {self EnsurePresent(L)}
      {@tupledict.L.insert Tuple}
    end
  end

  meth read(L ?Tuple)
    lock
      {self EnsurePresent(L)}
      {@tupledict.L.delete Tuple}
      {self Cleanup(L)}
    end
    {Wait Tuple}
  end

  meth readnonblock(L ?Tuple ?B)
    lock
      {self EnsurePresent(L)}
      if {@tupledict.L.size}>0 then
        {@tupledict.L.delete Tuple} B=true
      else B=false end
      {self Cleanup(L)}
    end
  end
end
end

```

Figure 8.12: Tuple space (object-oriented version)

```

fun {SimpleLock}
  Token={NewCell unit}
  proc {Lock P}
    Old New in
      {Exchange Token Old New}
      {Wait Old}
      {P}
      New=unit
  end
in
  `lock`(`lock`:Lock)
end

```

Figure 8.13: Lock (non-reentrant version without exception handling)

```

fun {CorrectSimpleLock}
  Token={NewCell unit}
  proc {Lock P}
    Old New in
      {Exchange Token Old New}
      {Wait Old}
      try {P} finally New=unit end
  end
in
  `lock`(`lock`:Lock)
end

```

Figure 8.14: Lock (non-reentrant version with exception handling)

invisible to users of tuple space objects (see Section 7.3.3). The implementation does correct memory management: a new entry is added upon the first occurrence of a particular label; and when the queue is empty, the entry is removed.

The tuple space implementation uses a concurrent stateful queue which is a slightly extended version of Figure 8.8. We add just one operation, a function that returns the size of the queue, i.e., the number of elements it contains. Our queue extends Figure 8.8 like this:

```

fun {NewQueue}
  ...
  fun {Size}
    lock L then @C.1 end
  end
in
  queue(insert:Insert delete>Delete size:Size)
end

```

We will extend this queue again for implementing monitors.

```

fun {NewLock}
  Token={NewCell unit}
  CurThr={NewCell unit}
  proc {Lock P}
    if {Thread.this}==@CurThr then
      {P}
    else Old New in
      {Exchange Token Old New}
      {Wait Old}
      CurThr:={Thread.this}
      try {P} finally
        CurThr:=unit
        New=unit
      end
    end
  end
in
  `lock`(`lock`:Lock)
end

```

Figure 8.15: Lock (reentrant version with exception handling)

8.3.3 Implementing locks

Locks can be defined in the concurrent stateful model by using cells and dataflow variables. We first show the definition of a simple lock, then a simple lock that handles exceptions correctly, and finally a thread-reentrant lock. The built-in locks provided by the system are thread-reentrant locks with the semantics given here, but they have a more efficient low-level implementation.

A *simple lock* is a procedure $\{L\ P\}$ that takes a zero-argument procedure P as argument and executes P in a critical section. Any thread that attempts to enter the lock while there is still one thread inside will suspend. The lock is called *simple* because a thread that is inside a critical section cannot enter any other critical section protected by the same lock. It first has to leave the initial critical section. Simple locks can be created by the function `SimpleLock` defined in Figure 8.13. If multiple threads attempt to access the lock body, then only one is given access and the others are queued. When a thread leaves the critical section, access is granted to the next thread in the queue. This uses the token passing technique of in Section 8.2.2.

But what happens if the lock body $\{P\}$ raises an exception? The lock of Figure 8.13 does not work since `New` will never be bound. We can fix this problem with a `try` statement. Figure 8.14 gives a version of the simple lock that handles exceptions. The `try <stmt>1 finally <stmt>2 end` is syntactic sugar that ensures $\langle \text{stmt} \rangle_2$ is executed in both the normal and exceptional cases, i.e., an exception will not prevent the lock from being released.

A *thread-reentrant* lock extends the simple lock to allow the same thread to enter other critical sections protected by the same lock. It is even possible to nest critical sections protected by the same lock. Other threads trying to acquire the lock will queue until `P` is completed. When the lock is released, it is granted to the thread standing first in line. Figure 8.15 shows how to define thread-reentrant locks. This assumes that each thread has a unique identifier `T` that is different from the literal `unit` and that is obtained by calling the procedure `{Thread.this T}`. The assignments to `CurThr` have to be done in exactly the places shown. What can go wrong if `{Wait Old}` and `CurThr := {Thread.this}` are switched or if `CurThr := unit` and `New = unit` are switched?

8.4 Monitors

Locks are an important tool for building concurrent abstractions in a stateful model, but they are not sufficient. For example, consider the simple case of a bounded buffer. A thread may want to put an element in the buffer. It is not enough to protect the buffer with a lock. What if the buffer is full: the thread enters and can do nothing! What we really want is a way for the thread to wait until the buffer is not full, and then continue. This cannot be done with just locks. It needs a way for threads to coordinate among each other. For example, a thread that puts an element in the buffer can be notified that the buffer is not full by another thread which removes an element from the buffer.

The standard way for coordinating threads in a stateful model is by using monitors. Monitors were introduced by Brinch Hansen [22, 23] and further developed by Hoare [82]. They continue to be widely used; for example they are a basic concept in the Java language [111]. A *monitor* is a lock extended with program control over how waiting threads enter and exit the lock. This control makes it possible to use the monitor as a resource that is shared among concurrent activities. There are several ways to give this control. Typically, a monitor has either one set of waiting threads or several queues of waiting threads. The simplest case is when there is one set; let us consider it first.

The monitor adds a `wait` and a `notify` operation to the lock entry and exit operations. (`notify` is sometimes called `signal`.) The `wait` and `notify` are only possible from inside the monitor. When inside a monitor, a thread can explicitly do a `wait`; thereupon the thread suspends, is entered in the monitor's wait set, and releases the monitor lock. When a thread does a `notify`, it lets one thread in the wait set continue. This thread attempts to get the monitor lock again. If it succeeds, it continues running from where it left off.

We first give an informal definition of monitors. We then program some examples both with monitors and in the declarative concurrent model. This will let us compare both approaches. We conclude the section by giving an implementation of monitors in the shared-state concurrent model.

Definition

There exist several varieties of monitors, with slightly different semantics. We first explain the Java version because it is simple and popular. (Section 8.4.4 gives an alternative version.) The following definition is taken from [110]. In Java, a monitor is always part of an object. It is an object with an internal lock and wait set. Object methods can be protected by the lock by annotating them as `synchronized`. There are three operations to manage the lock: `wait`, `notify`, and `notifyAll`. These operations can only be called by threads that hold the lock. They have the following meaning:

- The `wait` operation does the following:
 - The current thread is suspended.
 - The thread is placed in the object's internal wait set.
 - The lock for the object is released.
- The `notify` operation does the following:
 - If one exists, an arbitrary thread `T` is removed from the object's internal wait set.
 - `T` proceeds to get the lock, just as any other thread. This means that `T` will always suspend for a short time, until the notifying thread releases the lock.
 - `T` resumes execution at the point it was suspended.
- The `notifyAll` operation is similar to `notify` except that it does the above steps for *all* threads in the internal wait set. The wait set is then emptied.

For the examples that follow, we suppose that a function `NewMonitor` exists with the following specification:

- `M={NewMonitor}` creates a monitor with operations `{M.ˆlockˆ}` (monitor lock procedure), `{M.wait}` (wait operation), `{M.notify}` (notify operation), and `{M.notifyAll}` (notifyAll operation).

In the same way as for locks, we assume that the monitor lock is thread-reentrant and handles exceptions correctly. Section 8.4.3 explains how the monitor is implemented.

Monitors were designed for building concurrent ADTs. To make it easier to build ADTs with monitors, some languages provide them as a linguistic abstraction. This makes it possible for the compiler to guarantee that the wait and notify operations are only executed inside the monitor lock. This can also make it easy for the compiler to ensure safety properties, e.g., that shared variables are only accessed through the monitor [24].

8.4.1 Bounded buffer

In Chapter 4, we showed how to implement a bounded buffer declaratively in two ways, with both eager and lazy stream communication. In this section we implement it with a monitor. We then compare this solution with the two declarative implementations. The bounded buffer is an ADT with three operations:

- $B = \{\text{New Buffer init}(N)\}$: create a new bounded buffer B of size N .
- $\{B \text{ put}(X)\}$: put the element x in the buffer. If the buffer is full, this will block until the buffer has room for the element.
- $\{B \text{ get}(X)\}$: remove the element x from the buffer. If the buffer is empty, this will block until there is at least one element.

The idea of the implementation is simple: the put and get operations will each wait until the buffer is not full and not empty, respectively. This gives the following partial definition:

```

class Buffer
  attr
    buf first last n i

  meth init(N)
    buf := {NewArray 0 N-1 null}
    first := 0 last := 0 n := N i := 0
  end

  meth put(X)
    ... % wait until i < n
    % now add an element:
    @buf.@last := X
    last := (@last + 1) mod @n
    i := i + 1
  end

  meth get(X)
    ... % wait until i > 0
    % now remove an element:
    X = @buf.@first
    first := (@first + 1) mod @n
    i := i - 1
  end
end

```

The buffer uses an array of n elements, indexed by `first` and `last`. The array wraps around: after element $n - 1$ comes element 0. The buffer's maximum size is n of which i elements are used. Now let's code it with a monitor. The naive solution is the following (where M is a monitor record):

```

meth put(X)
  {M.‘lock’ proc {$}
    if @i>=@n then {M.wait} end
    @buf.@last:=X
    last:=(@last+1) mod @n
    i:=@i+1
    {M.notifyAll}
  end}
end

```

That is, if the buffer is full, then {M.wait} simply waits until it is no longer full. When get(X) removes an element, it does a {M.notifyAll}, which wakes up the waiting thread. This naive solution is not good enough, since there is no guarantee that the buffer will not fill up just after the wait. When the thread releases the monitor lock with {M.wait}, other threads can slip in to add and remove elements. A correct solution does {M.wait} as often as necessary, checking the comparison @i>=@n each time. This gives the following code:

```

meth put(X)
  {M.‘lock’ proc {$}
    if @i>=@n then
      {M.wait}
      {self put(X)}
    else
      @buf.@last:=X
      last:=(@last+1) mod @n
      i:=@i+1
      {M.notifyAll}
    end
  end}
end

```

After the wait, this calls the put method again to do the check again. Since the lock is reentrant, it will let the thread enter again. The check is done inside the critical section, which eliminates any interference from other threads. Now we can put the pieces together. Figure 8.16 gives the final solution. The init method creates the monitor and stores the monitor procedures in object attributes. The put and get methods use the technique we gave above of waiting in a loop.

Let us compare this version with the declarative concurrent versions of Chapter 4. Figure 4.15 gives the eager version and Figure 4.28 gives the lazy version. The lazy version is the simplest. Either of the declarative concurrent versions can be used whenever there is no observable nondeterminism, for example, in point-to-point connections to connect one writer with one reader. Another case is when there are multiple readers that all read the same items. The monitor version can be used when the number of independent writers is more than one or when the number of independent readers is more than one.


```

class Buffer
  attr m buf first last n i

  meth init(N)
    m:={NewMonitor}
    buf:={NewArray 0 N-1 null}
    n:=N i:=0 first:=0 last:=0
  end

  meth put(X)
    {@m.´lock´ proc {$}
      if @i>=@n then
        {@m.wait}
        {self put(X)}
      else
        @buf.@last:=X
        last:=(@last+1) mod @n
        i:=@i+1
        {@m.notifyAll}
      end
    end}
  end

  meth get(X)
    {@m.´lock´ proc {$}
      if @i==0 then
        {@m.wait}
        {self get(X)}
      else
        X=@buf.@first
        first:=(@first+1) mod @n
        i:=@i-1
        {@m.notifyAll}
      end
    end}
  end
end

```

Figure 8.16: Bounded buffer (monitor version)

8.4.2 Programming with monitors

The technique we used in the bounded buffer is a general one for programming with monitors. Let us explain it in the general setting. For simplicity, assume that we are defining a concurrent ADT completely in a single class. The idea is that each method is a critical section that is *guarded*, i.e., there is a boolean condition that must be true for a thread to enter the method body. If the condition is false, then the thread waits until it becomes true. A guarded method is also called a *conditional critical section*.

Guarded methods are implemented using the `wait` and `notifyAll` operations. Here is an example in a simple pseudocode:

```
meth methHead
  lock
    while not <expr> do wait;
    <stmt>
    notifyAll;
  end
end
```

In this example, `<expr>` is the guard and `<stmt>` is the guarded body. When the method is called, the thread enters the lock and waits for the condition in a `while` loop. If the condition is true then it immediately executes the body. If the condition is false then it waits. When the wait continues then the loop is repeated, i.e., the condition is checked again. This guarantees that the condition is true when the body is executed. Just before exiting, the method notifies all other waiting threads that they might be able to continue. They will all wake up and try to enter the monitor lock to test their condition. The first one that finds a true condition is able to continue. The others will wait again.

8.4.3 Implementing monitors

Let us show how to implement monitors in the shared-state concurrent model. This gives them a precise semantics. Figure 8.19 shows the implementation. It is thread-reentrant and correctly handles exceptions. It implements mutual exclusion using the get-release lock of Figure 8.18. It implements the wait set using the extended queue of Figure 8.17. Implementing the wait set with a queue avoids starvation because it gives the longest-waiting thread the first chance to enter the monitor.

The implementation only works if `M.wait` is always executed inside an active lock. To be practical, the implementation should be extended to check this at run-time. We leave this simple extension up to the reader. Another approach is to embed the implementation inside a linguistic abstraction that statically enforces this.

When writing concurrent programs in the shared-state concurrent model, it is usually simpler to use the dataflow approach rather than monitors. The Mozart

```

fun {NewQueue}
  ...
  fun {Size}
    lock L then @C.1 end
  end
  fun {DeleteAll}
    lock L then
      X q(_ S E)=@C in
        C:=q(0 X X)
        E=nil S
    end
  end
  fun {DeleteNonBlock}
    lock L then
      if {Size}>0 then [{Delete}] else nil end
    end
  in
    queue(insert:Insert delete:Delete size:Size
          deleteAll:DeleteAll deleteNonBlock:DeleteNonBlock)
end

```

Figure 8.17: Queue (extended concurrent stateful version)

implementation therefore does no special optimizations to improve monitor performance. However, the implementation of Figure 8.19 can be optimized in many ways, which is important if monitor operations are frequent.

Extended concurrent queue

For the monitor implementation, we extend the concurrent queue of Figure 8.8 with the three operations `Size`, `DeleteAll`, and `DeleteNonBlock`. This gives the definition of Figure 8.17.

This queue is a good example of why reentrant locking is useful. Just look at the definition of `DeleteNonBlock`: it calls `Size` and `Delete`. This will only work if the lock is reentrant.

Reentrant get-release lock

For the monitor implementation, we extend the reentrant lock of Figure 8.15 to a get-release lock. This exports the actions of getting and releasing the lock as separate operations, `Get` and `Release`. This gives the definition of Figure 8.18. The operations have to be separate because they are used in both `LockM` and `WaitM`.

```

fun {NewGRLock}
  Token1={NewCell unit}
  Token2={NewCell unit}
  CurThr={NewCell unit}

  proc {GetLock}
    if {Thread.this}\=@CurThr then Old New in
      {Exchange Token1 Old New}
      {Wait Old}
      Token2:=New
      CurThr:={Thread.this}
    end
  end

  proc {ReleaseLock}
    CurThr:=unit
    unit=@Token2
  end
in
  `lock`(get:GetLock release:ReleaseLock)
end

```

Figure 8.18: Lock (reentrant get-release version)

8.4.4 Another semantics for monitors

In the monitor concept we introduced above, **notify** has just one effect: it causes one waiting thread to leave the wait set. This thread then tries to obtain the monitor lock. The notifying thread does not immediately release the monitor lock. When it does, the notified thread competes with other threads for the lock. This means that an assertion satisfied at the time of the notify might no longer be satisfied when the notified thread enters the lock. This is why an entering thread has to check the condition again.

There is a variation that is both more efficient and easier to reason about. It is for **notify** to do two operations atomically: it first causes one waiting thread to leave the wait set (as before) and it then immediately passes the monitor lock to that thread. The notifying thread thereby exits from the monitor. This has the advantage that an assertion satisfied at the time of the notify will still be true when the notified thread continues. The **notifyAll** operation no longer makes any sense in this variation, so it is left out.

Languages that implement monitors this way usually allow to declare several wait sets. A wait set is seen by the programmer as an instance of a special ADT called a *condition*. The programmer can create new instances of conditions, which are called condition variables. Each condition variable *c* has two operations, *c.wait* and *c.notify*.

```

fun {NewMonitor}
  Q={NewQueue}
  L={NewGRLock}

  proc {LockM P}
    {L.get} try {P} finally {L.release} end
  end

  proc {WaitM}
  X in
    {Q.insert X} {L.release} {Wait X} {L.get}
  end

  proc {NotifyM}
  U={Q.deleteNonBlock} in
    case U of [X] then X=unit else skip end
  end

  proc {NotifyAllM}
  L={Q.deleteAll} in
    for X in L do X=unit end
  end
in
  monitor(`lock`:LockM wait:WaitM notify:NotifyM
        notifyAll:NotifyAllM)
end

```

Figure 8.19: Monitor implementation

We can reimplement the bounded buffer using this variation. The new bounded buffer has two conditions, which we can call *nonempty* and *nonfull*. The *put* method waits for a *nonfull* and then signals a *nonempty*. The *get* method waits for a *nonempty* and then signals a *nonfull*. This is more efficient than the previous implementation because it is more selective. Instead of waking up all the monitor's waiting threads with *notifyAll*, only one thread is woken up, in the right wait set. We leave the actual coding to an exercise.

8.5 Transactions

Transactions were introduced as a basic concept for the management of large shared databases. Ideally, databases must sustain a high rate of concurrent updates while keeping the data coherent and surviving system crashes. This is not an easy problem to solve. To see why, consider a database represented as a large array of cells. Many clients wish to update the database concurrently. A naive implementation is to use a single lock to protect the whole array. This solution is

impractical for many reasons. One problem is that a client that takes one minute to perform an operation will prevent any other operation from taking place during that time. This problem can be solved with transactions.

The term “transaction” has acquired a fairly precise meaning: it is any operation that satisfies the four ACID properties [16, 64]. ACID is an acronym:

- A stands for *atomic*: no intermediate states of a transaction’s execution are observable. It is as if the transaction happened instantaneously or did not happen at all. The transaction can complete normally (it *commits*) or it can be canceled (it *aborts*).
- C stands for *consistent*: observable state changes respect the system invariants. Consistency is closely related to atomicity. The difference is that consistency is the responsibility of the programmer, whereas atomicity is the responsibility of the implementation of the transaction system.
- I stands for *isolation*: several transactions can execute concurrently without interfering with each other. They execute *as if* they were sequential. This property is also called *serializability*. It means that the transactions have an interleaving semantics, just like the underlying computation model. We have “lifted” the interleaving semantics up from the model to the level of the transactions.
- D stands for *durability*: observable state changes survive across system shutdowns. Durability is often called *persistence*. Implementing durability requires a stable storage (such as a disk) that stores the observable state changes.

This chapter only gives a brief introduction to transaction systems. The classic reference on transactions is Bernstein *et al* [16]. This book is clear and precise and introduces the theory of transactions with just the right amount of formalism to aid intuition. Unfortunately, this book is out of print. Good libraries will often have a copy. Another good book on transactions is Gray & Reuter [64]. An extensive and mathematically rigorous treatment is given by Weikum & Vossen [204].

Lightweight (ACI) transactions

Outside of database applications all four ACID properties are not always needed. This section uses the term “transaction” in a narrower sense that is closer to the needs of general-purpose concurrent programming. Whenever there is a risk of confusion, we will call it a light transaction. A *lightweight transaction* is simply an abortable atomic action. It has all ACID properties except for D (durability). A lightweight transaction can commit or abort. The abort can be due to a cause internal to the program (e.g., because of conflicting access to shared data) or external to the program (e.g., due to failure of part of the system, like a disk or the network).

Motivations

We saw that one motivation for transactions was to increase the throughput of concurrent accesses to a database. Let us look at some other motivations. A second motivation is concurrent programming with exceptions. Most routines have two possible ways to exit: either they exit normally or they raise an exception. Usually the routine behaves atomically when it exits normally, i.e., the caller sees the initial state and the result but nothing in between. When there is an exception this is not the case. The routine might have put part of the system in an inconsistent state. How can we avoid this undesirable situation? There are two solutions:

- The caller can clean up the called routine's mess. This means that the called routine has to be carefully written so that its mess is always limited in extent.
- The routine can be inside a transaction. This solution is harder to implement, but can make the program much simpler. Raising an exception corresponds to aborting the transaction.

A third motivation is fault tolerance. Lightweight transactions are important for writing fault-tolerant applications. When a fault occurs, a fault-tolerant application has to take three steps: (1) detect the fault, (2) contain the fault in a limited part of the application, and (3) handle the fault. Lightweight transactions are a good mechanism for fault confinement.

A fourth motivation is resource management. Lightweight transactions allow to acquire multiple resources without causing a concurrent application to stop because of an undesirable situation called deadlock. This situation is explained below.

Kernel language viewpoint Let us make a brief detour and examine transactions from the viewpoint of computation models. The transactional solution satisfies one of our criteria for adding a concept to the computation model, namely that programs in the extended model are simpler. But what exactly is the concept to be added? This is still an open research subject. In our view, it is a very important one. Some day, the solution to this question will be an important part of all general-purpose programming languages. In this section we do not solve this problem. We will implement transactions as an abstraction in the concurrent stateful model without changing the model.

8.5.1 Concurrency control

Consider a large database accessed by many clients at the same time. What does this imply for transactions? It means that they are concurrent yet still satisfy serializability. The implementation should allow concurrent transactions and yet

it has to make sure that they are still serializable. There is a strong tension between these two requirements. They are not easy to satisfy simultaneously. The design of transaction systems that satisfy both has led to a rich theory and many clever algorithms [16, 64].

Concurrency control is the set of techniques used to build and program concurrent systems with transactional properties. We introduce these techniques and the concepts they are based on and we show one practical algorithm. Technically speaking, our algorithm does optimistic concurrency control with strict two-phase locking and deadlock avoidance. We explain what all these terms mean and why they are important. Our algorithm is interesting because it is both practical and simple. A complete working implementation takes just two pages of code.

Locks and timestamps

The two most widely-used approaches to concurrency control are locks and timestamps:

- *Lock-based concurrency control.* Each stateful entity has a lock that controls access to the entity. For example, a cell might have a lock that permits only one transaction to use it at a time. In order to use a cell, the transaction must have a lock on it. Locks are important to enforce serializability. This is a *safety property*, i.e., an assertion that is always true during execution. A safety property is simply a system invariant. In general, locks allow to restrict the system's behavior so that it is safe.
- *Timestamp-based concurrency control.* Each transaction is given a timestamp that gives it a priority. The timestamps are taken from an ordered set, something like the numbered tickets used in shops to ensure that customers are served in order. Timestamps are important to ensure that execution makes progress. For example, that each transaction will eventually commit or abort. This is a *liveness property*, i.e., an assertion that always eventually becomes true.

Safety and liveness properties describe how a system behaves as a function of time. To reason with these properties, it is important to be careful about the exact meanings of the terms “is always true” and “eventually becomes true”. These terms are relative to the current execution step. A property *is always true* if it is true at every execution step starting from the current step. A property *eventually becomes true* if there exists at least one execution step in the future where it is true. We can combine always and eventually to make more complicated properties. For example, a property that *always eventually becomes true* means that at every step starting from the current step it will eventually become true. The property “an active transaction will eventually abort or commit” is of this type. This style of reasoning can be given a formal syntax and semantics. This results in a variety of logic called *temporal logic*.

Optimistic and pessimistic scheduling

There are many algorithms for concurrency control, which vary on different axes. One of these axes is the degree of optimism or pessimism of the algorithm. Let us introduce this with two examples taken from real life. Both examples concern traveling, by airplane or by train.

Airlines often overbook flights, that is, sell more tickets than there is room on the flight. At boarding time, there have usually been enough cancellations that this is not a problem (all passengers have a seat). But occasionally some passengers have no seat, and these have to be accommodated in some way (e.g., by booking them on a later flight and reimbursing their discomfort). This is an example of *optimistic scheduling*: a passenger requesting a ticket is given the ticket right away even if the flight is already completely booked, as long as the overbooking is less than some ratio. Occasional problems are tolerated since overbooking allows to increase the average number of filled seats on a flight and because problems are easily repaired.

Railways are careful to ensure that there are never two trains traveling towards each other on the same track segment. A train is only allowed to enter a track segment if at that moment there is no other train on the same segment. Protocols and signaling mechanisms have been devised to ensure this. This is an example of *pessimistic scheduling*: a train requesting to enter a segment may have to wait until the segment is known to be clear. Unlike the case of overbookings, accidents are not tolerated because they are extremely costly and usually irreparable in terms of people's lives lost.

Let us see how these approaches apply to transactions. A transaction requests a lock on a cell. This request is given to a *scheduler*. The scheduler decides when and if the request should be fulfilled. It has three possible responses: to satisfy the request immediately, to reject the request (causing a transaction abort), or to postpone its decision. An optimistic scheduler tends to give the lock right away, even if this might cause problems later on (deadlocks and livelocks, see below). A pessimistic scheduler tends to delay giving the lock, until it is sure that no problems can occur. Depending on how often transactions work on shared data, an optimistic or pessimistic scheduler might be more appropriate. For example, if transactions mostly work on independent data, then an optimistic scheduler may give higher performance. If transactions often work on shared data, then a pessimistic scheduler may give higher performance. The algorithm we give below is an optimistic one; it sometimes has to repair mistakes due to past choices.

Two-phase locking

Two-phase locking is the most popular technique for doing locking. It is used by almost all commercial transaction processing systems. It can be proved that doing two-phase locking guarantees that transactions are serializable. In two-phase locking a transaction has two phases: a *growing* phase, in which it acquires locks but does not release them, and a *shrinking* phase, in which it releases locks

but does not acquire them. A transaction is not allowed to release a lock and then acquire another lock afterwards. This restriction means that a transaction might hold a lock longer than it needs to. Experience shows that this is not a serious problem.

A popular refinement of two-phase locking used by many systems is called *strict* two-phase locking. In this refinement, all locks are released simultaneously at the end of the transaction, after it commits or aborts. This avoids a problem called *cascading abort*. Consider the following scenario. Assume that standard two-phase locking is used with two transactions T1 and T2 that share cell C. First T1 locks C and changes C's content. Then T1 releases the lock in its shrinking phase but continues to be active. Finally T2 locks C, does a calculation with C, and commits. What happens if T1, which is still active, now aborts? If T1 aborts then T2 has to abort too, since it has read a value of C modified by T1. T2 could be linked in a similar way to another transaction T3, and so forth. If T1 aborts then all the others have to abort as well, in cascade, even though they already committed. If locks are released only after transactions commit or abort then this problem does not occur.

8.5.2 A simple transaction manager

Let us design a simple transaction manager. It will do optimistic concurrency control with strict two-phase locking. We first design the algorithm using stepwise refinement. We then show how to implement a transaction manager that is based on this algorithm.

A naive algorithm

We start the design with the following simple idea. Whenever a transaction requests the lock of an unlocked cell, let it acquire the lock immediately without any further conditions. If the cell is already locked, then let the transaction wait until it becomes unlocked. When a transaction commits or aborts, then it releases all its locks. This algorithm is optimistic because it assumes that getting the lock will not give problems later on. If problems arise (see next paragraph!) then the algorithm has to fix them.

Deadlock

Our naive algorithm has a major problem: it suffers from deadlocks. Consider two concurrent transactions T1 and T2 where each one uses cells C1 and C2. Let transaction T1 use C1 and C2, in that order, and transaction T2 use C2 and C1, in the reverse order. Because of concurrency, it can happen that T1 has C1's lock and T2 has C2's lock. When each transaction tries to acquire the other lock it needs, it waits. Both transactions will therefore wait indefinitely. This kind of situation, in which active entities (transactions) wait for resources (cells) in a cycle, such that no entity can continue, is called a *deadlock*.

How can we ensure that our system never suffers from the consequences of deadlock? Like for ailments in general, there are two basic approaches: prevention and cure. The goal of *deadlock prevention* (also called *avoidance*) is to prevent a deadlock from ever happening. A transaction is prevented from locking an object that might lead to a deadlock. The goal of *deadlock cure* (also called *detection & resolution*) is to detect when a deadlock occurs and to take some action to reverse its effects.

Both approaches are based on a concept called the *wait-for graph*. This is a directed graph that has nodes for active entities (e.g., transactions) and resources (e.g., cells). There is an edge from each active entity to the resource it is waiting for (if any) but does not yet have. There is an edge from each resource to the active entity (if any) that has it. A deadlock corresponds to a cycle in the wait-for graph. Deadlock avoidance forbids adding an edge that would make a cycle. Deadlock detection detects the existence of a cycle and then removes one of its edges. The algorithm we give below does deadlock avoidance. It keeps a transaction from getting a lock that might result in a deadlock.

The correct algorithm

We can avoid deadlocks in the naive algorithm by giving earlier transactions higher priority than later transactions. The basic idea is simple. When a transaction tries to acquire a lock, it compares its priority with the priority of the transaction already holding the lock. If the latter has lower priority, i.e., it is a more recent transaction, then it is restarted and the former gets the lock. Let us define an algorithm based on this idea. We assume that transactions perform operations on cells and that each cell comes with a priority queue of waiting transactions, i.e., the transactions wait in order of their priorities. We use timestamps to implement the priorities. Here is the complete algorithm:

- A new transaction is given a priority that is lower than all active transactions.
- When a transaction tries to acquire a lock on a cell, then it does one of the following:
 - If the cell is currently unlocked, then the transaction immediately takes the lock and continues.
 - If the cell is already locked by the transaction, then the transaction just continues.
 - If the cell is locked by a transaction with higher priority, then the current transaction waits, i.e., it enqueues itself on the cell's queue.
 - If the cell is locked by a transaction with lower priority, then restart the latter and give the lock to the transaction with higher priority. A *restart* consists of two actions: first to abort the transaction and second to start it again with the same priority.

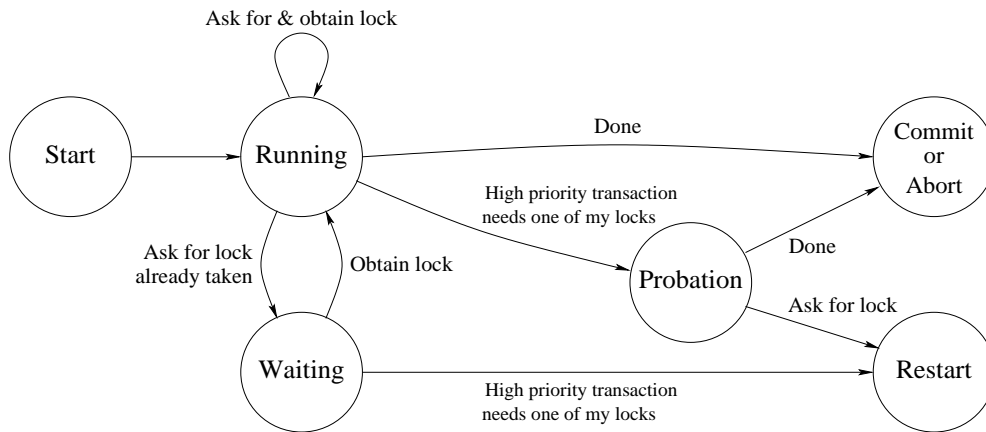


Figure 8.20: State diagram of one incarnation of a transaction

- When a transaction commits, then it releases all its locks and dequeues one waiting transaction per released lock (if there is one waiting).
- When a transaction aborts (because it raises an exception or explicitly does an abort operation) then it unlocks all its locked cells, restores their states, and dequeues one waiting transaction per unlocked cell (if there is one waiting).

Restarting at a well-defined point

There is a small problem with the above algorithm. It terminates running transactions at an *arbitrary point* during their execution. This can give problems. It can lead to inconsistencies in the run-time data structures of the transaction. It can lead to complications in the implementation of the transaction manager itself.

A simple solution to these problems is to terminate the transaction at a *well-defined point* in its execution. A well-defined point is, for example, the instant when a transaction asks the transaction manager for a lock. Let us refine the above algorithm to restart only at such points. Again, we start with a simple basic idea: instead of restarting a low priority transaction, we *mark* it. Later, when it tries to acquire a lock, the transaction manager notices that it is marked and restarts it. To implement this idea, we extend the algorithm as follows:

- Transactions can be in one of three states (the *marks*):
 - **running**: this is the unmarked state. The transaction is running freely and is able to acquire locks as before.
 - **probation**: this is the marked state. The transaction still runs freely, but the next time it tries to acquire a lock, it will be restarted. If it asks for no more locks, it will eventually commit.

- `waiting_on(C)`: this means that the transaction is waiting for the lock on cell `C`. It will obtain the lock when it becomes available. However, if a high priority transaction wants a lock held by this one while it is waiting, it will be restarted.

Figure 8.20 gives the state diagram of one incarnation of a transaction according to this scheme. By *incarnation* we mean part of the lifetime of a transaction, from its initial start or a restart until it commits, aborts, or again restarts.

- When a transaction tries to acquire a lock, then it checks its state before attempting to acquire locks. If it is in the state `probation` then it is restarted immediately. This is fine, since the transaction is at a well-defined point.
- When a transaction tries to acquire a lock and the cell is locked by a transaction with lower priority, then do the following. Enqueue the high priority transaction and take action depending on the state of the low priority transaction:
 - `running`: change the state to `probation` and continue.
 - `probation`: do nothing.
 - `waiting_on(C)`: remove the low priority transaction from the queue it is waiting on and restart it immediately. This is fine, since it is at a well-defined point.
- When a transaction is enqueued on a cell `C`, change its state to `waiting_on(C)`. When a transaction is dequeued, change its state to `running`.

8.5.3 Transactions on cells

Let us define an ADT for doing transactions on cells that uses the algorithm of the previous section.³ We define the ADT as follows:

- `{NewTrans ?Trans ?NewCellT}` creates a new transaction context and returns two operations: `Trans` for creating transactions and `NewCellT` for creating new cells.
- A new cell is created by calling `NewCellT` in the same way as with the standard `NewCell`:

`{NewCellT X C}`

³A similar ADT can be defined for objects, but the implementation is a little more complicated since we have to take into account classes and methods. For simplicity we will therefore limit ourselves to cells.

This creates a new cell in the transaction context and binds it to `c`. The cell can only be used inside transactions of this context. The initial value of the cell is `x`.

- A new transaction is created by calling the function `Trans` as follows:

```
{Trans fun {$ T} <expr> end B}
```

The sequential expression `<expr>` can interact with its environment in only the following ways: it can read values (including procedures and functions) and it can perform operations on cells created with `NewCellT`. The `Trans` call executes `<expr>` in a transactional manner and completes when `<expr>` completes. If `<expr>` raises an exception then the transaction will abort and raise the same exception. If the transaction commits, then it has the same effect as an atomic execution of `<expr>` and it returns the same result. If the transaction aborts, then it is as if `<expr>` were not executed at all (all its state changes are undone). `B` is bound to `commit` or `abort`, respectively, depending on whether the transaction commits or aborts.

- There are four operations that can be performed inside `<expr>`:
 - `T.access`, `T.assign`, and `T.exchange` have the same semantics as the standard three cell operations. They must only use cells created by `NewCellT`.
 - `T.abort` is a zero-argument procedure that when called causes the transaction to abort immediately.
- There are only two ways a transaction can abort: either it raises an exception or it calls `T.abort`. In all other cases, the transaction will eventually commit.

An example

Let us first create a new transaction environment:

```
declare Trans NewCellT in
{NewTrans Trans NewCellT}
```

We first define two cells in this environment:

```
C1={NewCellT 0}
C2={NewCellT 0}
```

Now let us increment `C1` and decrement `C2` in the same transaction:

```
{Trans proc {$ T _}
  {T.assign C1 {T.access C1}+1}
  {T.assign C2 {T.access C2}-1}
end _ _}
```