Beautiful concurrency

to appear in “Beautiful code”, ed Greg Wilson, O’Reilly 2007

Simon Peyton Jones, Microsoft Research, Cambridge

May 1, 2007

1 Introduction

The free lunch is over [11]. We have grown used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year’s model. If we want our programs to run faster, we must learn to write parallel programs [12].

Parallel programs execute in a non-deterministic way, so they are hard to test and bugs can be almost impossible to reproduce. For me, a beautiful program is one that is so simple and elegant that it obviously has no mistakes, rather than merely having no obvious mistakes[1]. If we want to write parallel programs that work reliably, we must pay particular attention to beauty. Sadly, parallel programs are often *less* beautiful than their sequential cousins; in particular they are, as we shall see, less *modular*.

In this chapter I’ll describe *Software Transactional Memory* (STM), a promising new approach to programming shared-memory parallel processors, that seems to support modular programs in a way that current technology does not. By the time we are done, I hope you will be as enthusiastic as I am about STM. It is not a solution to every problem, but it is a beautiful and inspiring attack on the daunting ramparts of concurrency.

2 A simple example: bank accounts

Here is a simple programming task.

> Write a procedure to transfer money from one bank account to another. To keep things simple, both accounts are held in memory: no

[1]This turn of phrase is due to Tony Hoare

> interaction with databases is required. The procedure must operate correctly in a concurrent program, in which many threads may call `transfer` simultaneously. No thread should be able to observe a state in which the money has left one account, but not arrived in the other (or vice versa).

This example is somewhat unrealistic, but its simplicity allows us to focus on what is new: the language Haskell (Section 3.1), and transactional memory (Sections 3.2 onwards). But first let us briefly look at the conventional approach.

2.1 Bank accounts using locks

The dominant technology for coordinating concurrent programs today is the use of *locks* and *condition variables*. In an object-oriented language, every object has an implicit lock and the locking is done by *synchronised methods*, but the idea is the same. So one might define a class for bank accounts something like this:

```
class Account {
  Int balance;
  synchronized void withdraw( int n ) {
    balance = balance - n; }
  void deposit( int n ) {
    withdraw( -n ); }
}
```

We must be careful to use a `synchronized` method for `withdraw`, so that we do not get any missed decrements if two threads call `withdraw` at the same time. The effect of `synchronized` is to take a lock on the account, run `withdraw`, and then release the lock.

Now, here is how we might write the code for `transfer`:

```
void transfer( Account from, Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount ); }
```

This code is fine for a sequential program, but in a concurrent program another thread could observe an intermediate state in which the money has left account `from`, but not arrived in `to`. The fact that both methods are `synchronized` does not help us at all. Account `from` is first locked and then unlocked by the call to method `withdraw`, and then `to` is locked and unlocked by `deposit`. In between the two calls, the money is (visibly) absent from both accounts.

In a finance program, that might be unacceptable. How do we fix it? The usual solution would be to add explicit locking code thus:

```
void transfer( Account from, Account to, Int amount ) {
```

```
from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
from.unlock(); to.unlock(); }
```

But this program is fatally prone to deadlock. In particular, consider the (unlikely) situation in which another thread is transferring money in the opposite direction between the same two accounts. Then each thread might get one lock and then block indefinitely waiting for the other.

Once recognised – and the problem is not always so obvious – the standard fix is to put an arbitrary global order on the locks, and to acquire them in increasing order. The locking code would then become

```
if from < to
  then { from.lock(); to.lock(); }
  else { to.lock(); from.lock(); }
```

That works fine when the full set of required locks can be predicted in advance, but that is not always the case. For example, suppose `from.withdraw` is implemented by transferring money out of account `from2` if `from` does not have enough funds. We don't know whether to acquire `from2`'s lock until we have read `from`, and by then it is too late to acquire the locks in the "right" order. Furthermore, the very existence of `from2` may be a private matter that should be known by `from`, but not by `transfer`. And even if `transfer` did know about `from2`, the locking code must now take three locks, presumably by sorting them into the right order.

Matters become even more complicated when we want to *block*. For example, suppose that `transfer` should block if `from` has insufficient funds. This is usually done by waiting on a *condition variable*, while simultaneously releasing `from`'s lock. It gets much trickier if we want to block until there are sufficient funds in `from` and `from2` considered together.

2.2 Locks are bad

To make a long story short, today's dominant technology for concurrent programming – locks and condition variables – is fundamentally flawed. Here are some standard difficulties, some of which we have seen above:

Taking too few locks. It is easy to forget to take a lock and thereby end up with two threads that modify the same variable simultaneously.

Taking too many locks. It is easy to take too many locks and thereby inhibit concurrency (at best) or cause deadlock (at worst).

Taking the wrong locks. In lock-based programming, the connection between a lock and the data it protects often exists only in the mind of the programmer, and is not explicit in the program. As a result, it is all too easy to take or hold the wrong locks.

Taking locks in the wrong order. In lock-based programming, one must be careful to take locks in the "right" order. Avoiding the deadlock that can otherwise occur is always tiresome and error-prone, and sometimes extremely difficult.

Error recovery can be very hard, because the programmer must guarantee that no error can leave the system in a state that is inconsistent, or in which locks are held indefinitely.

Lost wake-ups and erroneous retries. It is easy to forget to signal a condition variable on which a thread is waiting; or to re-test a condition after a wake-up.

But the fundamental shortcoming of lock-based programming is that *locks and condition variables do not support modular programming.* By "modular programming" I mean the process of building large programs by gluing together smaller programs. Locks make this impossible. For example, we could not use our (correct) implementations of `withdraw` and `deposit` unchanged to implement `transfer`; instead we had to expose the locking protocol. Blocking and choice are even less modular. For example suppose we had a version of `withdraw` that blocks if the source account has insufficient funds. Then we would not be able to use `withdraw` directly to withdraw money from A or B (depending on which has sufficient funds), without exposing the blocking condition — and even then it's not easy. This critique is elaborated elsewhere [7, 8, 4].

3 Software Transactional Memory

Software Transactional Memory is a promising new approach to the challenge of concurrency, as I will explain in this section. I shall explain STM using Haskell, the most beautiful programming language I know, because STM fits into Haskell particularly elegantly. If you don't know any Haskell, don't worry; we'll learn it as we go.

3.1 Side effects and input/output in Haskell

Here is the beginning of the code for `transfer` in Haskell:

```
transfer :: Account -> Account -> Int -> IO ()
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount = ...
```

The second line of this definition, starting "`--`", is a comment. The first line gives the *type signature* for `transfer`. This signature says that `transfer` takes as its arguments[2] two values of type `Account` (the source and destination accounts), and an `Int` (the amount to transfer), and returns a value of type `IO ()`. This result type says "`transfer` returns an action that, when performed, may have some side effects, and then returns a value of type `()`". The type `()`, pronounced "unit", has just one value, which is also written `()`; it is akin to `void` in C. So `transfer`'s result type `IO ()` announces that its side effects constitute the only reason for calling it. Before we go further, we must explain how side effects are handled in Haskell.

A "side effect" is anything that reads or writes mutable state. Input/output is a prominent example of a side effect. For example, here are the signatures of two Haskell functions with input/output effects:

```
hPutStr  :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
```

We call any value of type `IO t` an "action". So `(hPutStr h "hello")`[3] is an action that, when performed, will print `"hello"` on handle[4] `h` and return the unit value. Similarly, `(hGetLine h)` is an action that, when performed, will read a line of input from handle `h` and return it as a `String`. We can glue together little side-effecting programs to make bigger side-effecting programs using Haskell's "`do`" notation. For example, `hEchoLine` reads a string from the input and prints it:

```
hEchoLine :: Handle -> IO String
hEchoLine h = do { s <- hGetLine h
                 ; hPutStr h ("I just read: " ++ s)
                 ; return s }
```

The notation `do {`a_1`; ... ;`a_n`}` constructs an action by gluing together the smaller actions $a_1 \dots a_n$ in sequence. So `hEchoLine h` is an action that, when performed, will first perform `hGetLine h` to read a line from `h`, naming the result `s`. Then it will perform `hPutStr` to print `s`, preceded[5] by "`I just read: `". Finally, it returns the string `s`. This last line is interesting, because `return` is not a built-in language construct: rather, it is a perfectly ordinary function with type

```
return :: a -> IO a
```

[2]You may think it odd that there are three function arrows in this type signature, rather than one. That's because Haskell supports *currying*, which you can find described in any book about Haskell (e.g. [13]), or on Wikipedia. For the purposes of this paper, simply treat all the types except the final one as arguments.

[3]In Haskell we write function application using simple juxtaposition. In most languages you would write "`hPutStr(h,"hello")`", but in Haskell you write simply `(hPutStr h "hello")`.

[4]A `Handle` in Haskell plays the role of a file descriptor in C: it says which file or pipe to read or write. As in Unix, there are three pre-defined handles, `stdin`, `stdout`, and `stderr`.

[5]The `(++)` operator concatenates two strings.

The action `return v`, when performed, returns `v` without having caused any side effects[6]. This function works on values of any type, and we indicate this by using a type variable `a` in its type.

Input/output is one important sort of side effect. Another is the act of reading or writing a mutable variable. For example, here is a function that increments the value of a mutable variable:

```
incRef :: IORef Int -> IO ()
incRef var = do { val <- readIORef var
                ; writeIORef var (val+1) }
```

Here, `incRef var` is an action that first performs `readIORef var` to read the value of the variable, naming its value `val`, and then performs `writeIORef` to write the value `(val+1)` into the variable. The types of `readIORef` and `writeIORef` are as follows:

```
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

A value of type `IORef t` should be thought of as a pointer, or reference, to a mutable location containing a value of type `t`, a bit like the type `(t *)` in C. In the case of `incRef`, the argument has type `IORef Int` because `incRef` only applies to locations that contain an `Int`.

So far I have explained how to build big actions by combining smaller ones together — but how does an action ever actually get performed? In Haskell, the whole program defines a single `IO` action, called `main`. To run the program is to perform the action `main`. For example, here is a complete program:

```
main :: IO ()
main = do { hPutStr stdout "Hello"
          ; hPutStr stdout " world\n" }
```

This program is a sequential program, because the `do`-notation combines `IO` actions in sequence. To construct a concurrent program we need one more primitive, `forkIO`:

```
forkIO :: IO a -> IO ThreadId
```

The function `forkIO`, which is built into Haskell, takes an `IO` action as its argument, and spawns it as a concurrent Haskell thread. Once created, it is run concurrently with all the other Haskell threads, by the Haskell runtime system. For example, suppose we modified our main program thus[7]:

```
main :: IO ()
main = do { forkIO (hPutStr stdout "Hello")
          ; hPutStr stdout " world\n" }
```

[6]The `IO` type indicates the *possibility* of side effects, not the certainty!

[7]In the first line of `main`, we could instead have written `tid <- forkIO (hPutStr ...)`, to bind the `ThreadId` returned by `forkIO` to `tid`. However, since we do not use the returned `ThreadId`, we are free to discard it by omitting the "`tid <-`" part.

Now the two `hPutStr` actions would run concurrently. Which of them would "win" (by printing its string first) is unspecified. Haskell threads spawned by `forkIO` are extremely lightweight: they occupy a few hundred bytes of memory, and it is perfectly reasonable for a single program to spawn thousands of them.

Gentle reader, you may by now be feeling that Haskell is a very clumsy and verbose language. After all, our three-line definition of `incRef` accomplishes no more than `x++` does in C! Indeed, in Haskell side effects are extremely explicit and somewhat verbose. However, remember first that Haskell is primarily a *functional* language. Most programs are written in the functional core of Haskell, which is rich, expressive, and concise. Haskell thereby gently encourages you to write programs that make sparing use of side effects.

Second, notice that being explicit about side effects reveals a good deal of useful information. Consider two functions:

```
f :: Int -> Int
g :: Int -> IO Int
```

From looking only at their types we can see that `f` is a pure function: it has no side effects. Given a particular `Int`, say `42`, the call `(f 42)` will return the same value every time it is called. In contrast, `g` has side effects, and this is apparent in its type. Each time `g` is performed it may give a different result — for example it may read from `stdin`, or modify a mutable variable — even if its argument is the same every time. This ability to make side effects explicit will prove very useful in what follows.

Lastly, actions are first-class values: they may be passed as arguments as well as returned as results. For example, here is the definition of a (simplified) `for` loop function, written entirely in Haskell rather than being built in:

```
nTimes :: Int -> IO () -> IO ()
nTimes 0 do_this = return ()
nTimes n do_this = do { do_this; nTimes (n-1) do_this }
```

This recursive function takes an `Int` saying how many times to loop, and an action `do_this`; it returns an action that, when performed, performs the `do_this` action `n` times. Here is an example of a use of `nTimes` to print "`Hello`" 10 times:

```
main = nTimes 10 (hPutStr stdout "Hello\n")
```

In effect, by treating actions as first-class values, Haskell supports *user-defined control structures.*

This chapter is not the place for a full introduction to Haskell, or even to side effects in Haskell. A good starting point for further reading is the tutorial *"Tackling the awkward squad"* [9].

3.2 Transactions in Haskell

Now we can return to our `transfer` function. Here is its code:

```
transfer :: Account -> Account -> Int -> IO ()
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount
  = atomically (do { deposit  to   amount
                   ; withdraw from amount })
```

The inner `do`-block should by now be fairly self-explanatory: we call `deposit` to deposit `amount` in `to`, and `withdraw` to withdraw `amount` from account `from`. We will write these auxiliary functions in a moment, but first look at the call to `atomically`. It takes an action as its argument, and performs it atomically. More precisely, it makes two guarantees:

Atomicity: the effects of `atomically act` become visible to another thread all at once. This ensures that no other thread can see a state in which money has been deposited in `to` but not yet withdrawn from `from`.

Isolation: during a call `atomically act`, the action `act` is completely unaffected by other threads. It is as if `act` takes a snapshot of the state of the world when it begins running, and then executes against that snapshot.

Here is a simple execution model for `atomically`. Suppose there is a single, global lock. Then `atomically act` grabs the lock, performs the action `act`, and releases the lock. This implementation brutally ensures that no two atomic blocks can be executed simultaneously, and thereby ensures atomicity.

There are two problems with this model. First, it does not ensure isolation at all: while one thread is accessing an `IORef` inside an atomic block (holding the Global Lock), there is nothing to stop *another* thread writing the same `IORef` directly (i.e. outside `atomically`, without holding the Global Lock), thereby destroying the isolation guarantee. Second, performance is dreadful, because every atomic block would be serialised even if no actual interference was possible.

I will discuss the second problem shortly, in Section 3.3. Meanwhile, the first objection is easily addressed with the type system. We give `atomically` the following type:

```
atomically :: STM a -> IO a
```

The argument of `atomically` is an action of type `STM a`. An `STM` action is like an `IO` action, in that it can have side effects, but the range of side effects for `STM` actions is much smaller. The main thing you can do in an `STM` action is read or write a transactional variable, of type `(TVar a)`, much as we could read or write `IORef`s in an `IO` action[8].

[8]The nomenclature is inconsistent here: it would be more consistent to use either `TVar` and `IOVar`, or `TRef` and `IORef`. But it would be disruptive to change at this stage; for better or worse we have `TVar` and `IORef`.

```
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

STM actions can be composed together with the same do-notation as IO actions — the do-notation is overloaded to work on both types, as is return[9]. Here, for example, is the code for withdraw:

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount
  = do { bal <- readTVar acc
       ; writeTVar acc (bal - amount) }
```

We represent an Account by a transactional variable containing an Int for the account balance. Then withdraw is an STM action that decrements the balance in the account by amount.

To complete the definition of transfer we can define deposit in terms of withdraw:

```
deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)
```

Notice that, transfer ultimately performs four primitive read/write actions: a read and then write on account to, followed by a read and then write on account from. These four actions execute atomically, and that meets the specification given at the start of Section 2.

The type system neatly prevents us from reading or writing a TVar outside of a transaction. For example, suppose we tried this:

```
bad :: Account -> IO ()
bad acc = do { hPutStr stdout "Withdrawing..."
             ; withdraw acc 10 }
```

This program is rejected because the hPutStr is an IO action, while the withdraw is an STM action, and the two cannot be combined in a single do block. If we wrap a call to atomically around the withdraw, all is well:

```
good :: Account -> IO ()
good acc = do { hPutStr stdout "Withdrawing..."
              ; atomically (withdraw acc 10) }
```

[9]This overloading of do-notation and return is not an *ad-hoc* trick to support IO and STM. Rather, IO and STM are both examples of a common pattern, called a *monad* [15], and the overloading is achieved by expressing that common pattern using Haskell's very general *type-class* mechanism [16, 10].

3.3 Implementing transactional memory

The guarantees of atomicity and isolation that I described earlier should be all that a programmer needs in order to use STM. Even so, I often find it helpful to have a reasonable implementation model to guide my intuitions, and I will sketch one such implementation in this section. But remember that this is just *one* possible implementation. One of the beauties of the STM abstraction is that it presents a small, clean interface that can be implemented in a variety of ways, some simple and some sophisticated.

One particularly attractive implementation is well established in the database world, namely *optimistic execution*. When `(atomically act)` is performed, a thread-local *transaction log* is allocated, initially empty. Then the action `act` is performed, without taking any locks at all. While performing `act`, each call to `writeTVar` writes the address of the `TVar` and its new value into the log; it does not write to the `TVar` itself. Each call to `readTVar` first searches the log (in case the `TVar` was written by an earlier call to `writeTVar`); if no such record is found, the value is read from the `TVar` itself, and the `TVar` and value read are recorded in the log. In the meantime, other threads might be running their own atomic blocks, reading and writing `TVar`s like crazy.

When the action `act` is finished, the implementation first *validates* the log and, if validation is successful, *commits* the log. The validation step examines each `readTVar` recorded in the log, and checks that the value in the log matches the value currently in the real `TVar`. If so, validation succeeds, and the commit step takes all the writes recorded in the log and writes them into the real `TVar`s.

These steps are performed truly indivisibly: the implementation disables interrupts, or uses locks or compare-and-swap instructions — whatever is necessary to ensure that validation and commit are perceived by other threads as completely indivisible. All of this is handled by the implementation, however, and the programmer does not need to know or care how it is done.

What if validation fails? Then the transaction has had an inconsistent view of memory. So we abort the transaction, re-initialise the log, and run `act` all over again. This process is called *re-execution*. Since none of `act`'s writes have been committed to memory, it is perfectly safe to run it again. However, notice that it is crucial that `act` contains no effects *other than* reads and writes on `TVar`s. For example, consider

```
  atomically (do { x <- readTVar xv
                 ; y <- readTVar yv
                 ; if x>y then launchMissiles
                          else return () })
```

where `launchMissiles :: IO ()` causes serious international side-effects. Since the atomic block is executed without taking locks, it might have an inconsistent view of memory if other threads are concurrently modifying `xv` and `yv`. If that happens, it would be a mistake to launch the missiles, and only *then* discover that validation fails so the transaction should be re-run. Fortunately, the type system prevents us running `IO` actions inside `STM` actions, so the above fragment would be rejected by the type checker. This is another big advantage of distinguishing the types of `IO` and `STM` actions.

3.4 Blocking and choice

Atomic blocks as we have introduced them so far are utterly inadequate to coordinate concurrent programs. They lack two key facilities: *blocking* and *choice*. In this section I'll describe how the basic STM interface is elaborated to include them in a fully-modular way.

Suppose that a thread should *block* if it attempts to overdraw an account (i.e. withdraw more than the current balance). Situations like this are common in concurrent programs: for example, a thread should block if it reads from an empty buffer, or when it waits for an event. We achieve this in STM by adding the single function `retry`, whose type is

```
retry :: STM a
```

Here is a modified version of `withdraw` that blocks if the balance would go negative:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; if amount > 0 && amount > bal
         then retry
         else writeTVar acc (bal - amount) }
```

The semantics of `retry` are simple: if a `retry` action is performed, the current transaction is abandoned and retried at some later time. It would be correct to retry the transaction immediately, but it would also be inefficient: the state of the account will probably be unchanged, so the transaction will again hit the `retry`. An efficient implementation would instead block the thread until some other thread writes to `acc`. How does the implementation know to wait on `acc`? Because the transaction read `acc` on the way to the `retry`, and that fact is conveniently recorded in the transaction log.

The conditional in `limitedWithdraw` has a very common pattern: check that a boolean condition is satisfied and, if not, `retry`. This pattern is easy to abstract as a function, `check`:

```
check :: Bool -> STM ()
check True  = return ()
check False = retry
```

Now we can use `check` to re-express `limitedWithdraw` a little more neatly:

```
atomically :: STM a -> IO a

retry  :: STM a
orElse :: STM a -> STM a -> STM a

newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Figure 1: The key operations of STM Haskell

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; check (amount <= 0 || amount <= bal)
       ; writeTVar acc (bal - amount) }
```

We now turn our attention to *choice*. Suppose you want to withdraw money from account A if it has enough money, but if not then withdraw it from account B? For that, we need the ability to choose an alternative action if the first one retries. To support choice, STM Haskell has one further primitive action, called `orElse`, whose type is

```
orElse :: STM a -> STM a -> STM a
```

Like `atomically` itself, `orElse` takes actions as its arguments, and glues them together to make a bigger action. Its semantics are as follows. The action `(orElse a1 a2)` first performs `a1`; if `a1` retries (i.e. calls `retry`), it tries `a2` instead; if `a2` also retries, the whole action retries. It may be easier to see how `orElse` is used:

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
  = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

Since the result of `orElse` is itself an `STM` action, you can feed it to another call to `orElse` and so choose among an arbitrary number of alternatives.

3.5 Summary so far

In this section I have introduced all the key transactional memory operations supported by STM Haskell. They are summarised in Figure 1. This figure includes one operation that has not so far arisen: `newTVar` is the way in which you can create new `TVar` cells, and we will use it in the following section.

4 The Santa Claus problem

I want to show you a complete, runnable concurrent program using STM. A well-known example is the so-called Santa Claus problem[10], originally due to Trono [14]:

> Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

Using a well-known example allows you to directly compare my solution with well-described solutions in other languages. In particular, Trono's paper gives a semaphore-based solution which is partially correct; Ben-Ari gives a solution in Ada95 and in Ada [1]; Benton gives a solution in Polyphonic $C^{\#}$ [2].

4.1 Reindeer and elves

The basic idea of the STM Haskell implementation is this. Santa makes one "`Group`" for the elves and one for the reindeer. Each elf (or reindeer) tries to join its `Group`. If it succeeds, it gets two "`Gates`" in return. The first `Gate` allows Santa to control when the elf can enter the study, and also lets Santa know when they are all inside. Similarly, the second `Gate` controls the elves leaving the study. Santa, for his part, waits for either of his two `Groups` to be ready, and then uses that `Group`'s `Gates` to marshal his helpers (elves or reindeer) through their task. Thus the helpers spend their lives in an infinite loop: try to join a group, move through the gates under Santa's control, and then delay for a random interval before trying to join a group again.

Rendering this informal description in Haskell gives the following code for an elf[11]:

```
elf1 :: Group -> Int -> IO ()
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group
                       ; passGate in_gate
                       ; meetInStudy elf_id
                       ; passGate out_gate }
```

[10]My choice was influenced by the fact that I am writing these words on 22 December.

[11]I have given this function a suffix "`1`" because it only deals with one iteration of the elf, whereas in reality the elves re-join the fun when they are done with their task. We will define `elf` in Section 4.3.

The `elf` is passed its `Group`, and an `Int` that specifies its elfin identity. This identity is used only in the call to `meetInStudy`, which simply prints out a message to say what is happening[12]

```
meetInStudy :: Int -> IO ()
meetInStudy id = putStr ("Elf " ++ show id ++ " meeting in the study\n")
```

The elf calls `joinGroup` to join its group, and `passGate` to pass through each of the gates:

```
joinGroup :: Group -> IO (Gate, Gate)
passGate  :: Gate  -> IO ()
```

The code for reindeer is identical, except that reindeer deliver toys rather than meeting in the study:

```
deliverToys :: Int -> IO ()
deliverToys id = putStr ("Reindeer " ++ show id ++ " delivering toys\n")
```

Since `IO` actions are first-class, we can abstract over the common pattern, like this:

```
helper1 :: Group -> IO () -> IO ()
helper1 group do_task = do { (in_gate, out_gate) <- joinGroup group
                           ; passGate in_gate
                           ; do_task
                           ; passGate out_gate }
```

The second argument of `helper1` is an `IO` action that is the helper's task, which the helper performs between the two `passGate` calls. Now we can specialise `helper1` to be either an elf or a reindeer:

```
elf1, reindeer1 :: Group -> Int -> IO ()
elf1      gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

4.2 Gates and Groups

The first abstraction is a `Gate`, which supports the following interface:

```
newGate     :: Int -> STM Gate
passGate    :: Gate -> IO ()
operateGate :: Gate -> IO ()
```

A `Gate` has a fixed *capacity*, n, which we specify when we make a new `Gate`, and a mutable *remaining capacity.* This remaining capacity is decremented whenever a helper calls `passGate` to go through the gate; if the remaining capacity is zero, `passGate` blocks. A `Gate` is created with zero remaining capacity, so that

[12]The function `putStr` is a library function that calls `hPutStr stdout`.

no helpers can pass through it. Santa opens the gate with `operateGate`, which sets its remaining capacity back to n.

Here, then, is a possible implementation of a `Gate`:

```
data Gate  = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

passGate :: Gate -> IO ()
passGate (MkGate n tv)
  = atomically (do { n_left <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ()
operateGate (MkGate n tv)
  = do { atomically (writeTVar tv n)
       ; atomically (do { n_left <- readTVar tv
                        ; check (n_left == 0) }) }
```

The first line declares `Gate` to be a new *data type*, with a single *data constructor* `MkGate`[13]. The constructor has two *fields*: an `Int` giving the gate capacity, and a `TVar` whose contents says how many helpers can go through the gate before it closes. If the `TVar` contains zero, the gate is closed.

The function `newGate` makes a new `Gate` by allocating a `TVar`, and building a `Gate` value by calling the `MkGate` constructor. Dually, `passGate` uses pattern-matching to take apart the `MkGate` constructor; then it decrements the contents of the `TVar`, using `check` to ensure there is still capacity in the gate, as we did with `withdraw` (Section 3.4). Finally, `operateGate` first opens the `Gate` by writing its full capacity into the `TVar`, and then waits for the `TVar` to be decremented to zero.

A `Group` has the following interface:

```
newGroup   :: Int -> IO Group
joinGroup  :: Group -> IO (Gate,Gate)
awaitGroup :: Group -> STM (Gate,Gate)
```

Again, a `Group` is created empty, with a specified capacity. An elf may join a group by calling `joinGroup`, a call that blocks if the group is full. Santa calls `awaitGroup` to wait for the group to be full; when it is full he gets the `Group`'s gates, *and* the `Group` is immediately re-initialised with fresh `Gate`s, so that another group of eager elves can start assembling.

Here is a possible implementation:

[13]A data type declaration is not unlike a C `struct` declaration, with `MkGate` being the structure tag.

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup n = atomically (do { g1 <- newGate n; g2 <- newGate n
                            ; tv <- newTVar (n, g1, g2)
                            ; return (MkGroup n tv) })
```

Again, `Group` is declared as a fresh data type, with constructor `MkGroup` and two fields: the `Group`'s full capacity, and a `TVar` containing its number of empty slots and its two `Gate`s. Creating a new `Group` is a matter of creating new `Gate`s, initialising a new `TVar`, and returning a structure built with `MkGroup`.

The implementations of `joinGroup` and `awaitGroup` are now more or less determined by these data structures:

```
joinGroup (MkGroup n tv)
  = atomically (do { (n_left, g1, g2) <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1, g1, g2)
                   ; return (g1,g2) })

awaitGroup (MkGroup n tv)
  = do { (n_left, g1, g2) <- readTVar tv
       ; check (n_left == 0)
       ; new_g1 <- newGate n; new_g2 <- newGate n
       ; writeTVar tv (n,new_g1,new_g2)
       ; return (g1,g2) }
```

Notice that `awaitGroup` makes new gates when it re-initialises the `Group`. This ensures that a new group can assemble while the old one is still talking to Santa in the study, with no danger of an elf from the new group overtaking a sleepy elf from the old one.

Reviewing this section, you may notice that I have given some of the `Group` and `Gate` operations `IO` types (e.g. `newGroup`, `joinGroup`), and some `STM` types (e.g. `newGate`, `awaitGroup`). How did I make these choices? For example, `newGroup` has an `IO` type, which means that I can never call it from within an `STM` action. But this is merely a matter of convenience: I could instead have given `newGroup` an `STM` type, by omitting the `atomically` in its definition. In exchange, I would have had to write `atomically (newGroup n)` at each call site, rather than merely `newGroup n`. The merit of giving `newGate` an `STM` type is that it is more composable, a generality that `newGroup` did not need in this program. In contrast, I wanted to call `newGate` inside `newGroup`, and so I gave `newGate` an `STM` type.

In general, when designing a library, you should give the functions `STM` types wherever possible. You can think of `STM` actions as Lego bricks that can be glued together, using `do {...}`, `retry`, and `orElse`, to make bigger `STM` actions. However, as soon as you wrap a block in `atomically`, making it an `IO` type, it can no longer be combined atomically with other actions. There is a good reason for that: a value of `IO` type can perform arbitrary, irrevocable input/output (such as `launchMissiles`).

It is therefore good library design to export `STM` actions (rather than `IO` actions) whenever possible, because they are composable; their type advertises that they do no irrevocable effects. The library client can readily get from `STM` to `IO` (using `atomically`), but not vice versa.

Sometimes, however, it is *essential* to use an `IO` action. Look at `operateGate`. The two calls to `atomically` cannot be combined into one, because the first has an externally-visible side effect (opening the gate), while the second blocks until all the elves have woken up and gone through it. So `operateGate` *must* have an `IO` type.

4.3 The main program

We will first implement the outer structure of the program, although we have not yet implemented Santa himself. Here it is.

```
main = do { elf_group <- newGroup 3
          ; sequence_ [ elf elf_group n | n <- [1..10] ]

          ; rein_group <- newGroup 9
          ; sequence_ [ reindeer rein_group n | n <- [1..9] ]

          ; forever (santa elf_group rein_group) }
```

The first line creates a `Group` for the elves with capacity 3. The second line is more mysterious: it uses a so-called *list comprehension* to create a list of `IO` actions and calls `sequence_` to execute them in sequence. The list comprehension `[`e`|`x`<-`xs`]` is read "the list of all e where x is drawn from the list xs". So the argument to `sequence_` is the list

```
[elf elf_group 1, elf elf_group 2, ..., elf elf_group 10]
```

Each of these calls yields an `IO` action that spawns an elf thread. The function `sequence_` takes a list of `IO` actions and returns an action that, when performed, runs each of the actions in the list in order[14]:

```
sequence_ :: [IO a] -> IO ()
```

An `elf` is built from `elf1`, but with two differences. First, we want the elf to loop indefinitely, and second, we want it to run in a separate thread:

[14]The type `[IO a]` means "a list of values of type `IO a`". You may also wonder about the underscore in the name `sequence_`: it's because there is a related function `sequence` whose type is `[IO a] -> IO [a]`, that gathers the results of the argument actions into a list. Both `sequence` and `sequence_` are defined in the `Prelude` library, which is imported by default.

```
elf :: Group -> Int -> IO ThreadId
elf gp id = forkIO (forever (do { elf1 gp id; randomDelay }))
```

The `forkIO` part spawns its argument as a separate Haskell thread (Section 3.1). In turn, `forkIO`'s argument is a call to `forever`, which runs *its* argument repeatedly (compare the definition of `nTimes` in Section 3.1):

```
forever :: IO () -> IO ()
-- Repeatedly perform the action
forever act = do { act; forever act }
```

Finally the expression `(elf1 gp id)` is an `IO` action, and we want to repeat that action indefinitely, followed each time by a random delay:

```
randomDelay :: IO ()
-- Delay for a random time between 1 and 1,000,000 microseconds
randomDelay = do { waitTime <- getStdRandom (randomR (1, 1000000))
                 ; threadDelay waitTime }
```

The rest of the main program should be self-explanatory. We make nine reindeer in the same way that we made ten elves, except that we call `reindeer` instead of `elf`:

```
reindeer :: Group -> Int -> IO ThreadId
reindeer gp id = forkIO (forever (do { reindeer1 gp id; randomDelay }))
```

The code for `main` finishes by re-using `forever` to run `santa` repeatedly. All that remains is to implement Santa himself.

4.4 Implementing Santa

Santa is the most interesting participant of this little drama, because he makes choices. He must wait until there is *either* a group of reindeer waiting, *or* a group of elves. Once he has made his choice of which group to attend to, he must take them through their task. Here is his code:

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
  = do { putStr "----------\n"

       ; (task, (in_gate, out_gate))
             <- atomically (orElse
                    (chooseGroup rein_gp "deliver toys")
                    (chooseGroup elf_gp  "meet in my study"))

       ; putStr ("Ho! Ho! Ho! let’s " ++ task ++ "\n")
       ; operateGate in_gate
             -- Now the helpers do their task
       ; operateGate out_gate }
```

```
    where
      chooseGroup :: Group -> String -> STM (String, (Gate,Gate))
      chooseGroup gp task = do { gates <- awaitGroup gp
                               ; return (task, gates) }
```

The choice is made by the `orElse`, which first attempts to choose the reindeer (thereby giving them priority), and otherwise choosing the elves. The `chooseGroup` function does an `awaitGroup` call on the appropriate group, and returns a pair consisting of a string indicating the task (delivering toys or meeting in the study) and the gates that Santa must operate to take the group through the task. Once the choice is made, Santa prints out a message and operates the two gates in sequence.

This implementation works fine, but we will also explore an alternative, more general version, because `santa` demonstrates a very common programming pattern. The pattern is this: a thread (Santa in this case) makes a choice in one atomic transaction, followed by one or more further consequential transactions. Another typical example might be: take a message from one of several message queues, act on the message, and repeat. In this case, the consequential action was very similar for elves and reindeer — in both cases, Santa had to print a message and operate two gates. But that would not work if Santa should do very different things for elves and reindeer. One approach would be to return a boolean indicating which was chosen, and dispatch on that boolean after the choice; but that becomes inconvenient as more alternatives are added. Here is another approach that works better:

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
  = do { putStr "----------\n"
       ; choose [(awaitGroup rein_gp, run "deliver toys"),
                 (awaitGroup elf_gp,  run "meet in my study")] }
  where
    run :: String -> (Gate,Gate) -> IO ()
    run task (in_gate,out_gate)
      = do { putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
           ; operateGate in_gate
           ; operateGate out_gate }
```

The function `choose` is like a guarded command: it takes a list of pairs, waits until the first component of a pair is ready to "fire", and then executes the second component. So `choose` has this type[15]:

```
choose :: [(STM a, a -> IO ())] -> IO ()
```

The guard is an `STM` action delivering a value of type `a`; when the `STM` action is ready (that is, does not retry), `choose` can pass the value to the second

[15]In Haskell, the type `[`*ty*`]` means a list whose elements have type *ty*. In this case `choose`'s argument is a list of pairs, written `(`ty_1`,`ty_2`)`; the first component of the pair has type `STM a`, while the second is a function with type `a->IO ()`.

component, which must therefore be a function expecting a value of type `a`. With this in mind, `santa` should be easy reading. He uses `awaitGroup` to wait for a ready `Group`; the `choose` function gets the pair of `Gate`s returned by `awaitGroup` and passes it to the `run` function. The latter operates the two gates in succession – recall that `operateGate` blocks until all the elves (or reindeer) have gone through the gate.

The code for `choose` is brief, but a little mind-bending:

```
choose :: [(STM a, a -> IO ())] -> IO ()
choose choices = do { act <- atomically (foldr1 orElse actions)
                    ; act }
  where
    actions :: [STM (IO ())]
    actions = [ do { val <- guard; return (rhs val) }
              | (guard, rhs) <- choices ]
```

First, it forms a list, `actions`, of `STM` actions, which it then combines with `orElse`. (The call `foldr1` $\oplus$ $[x_1, \ldots, x_n]$ returns $x_1 \oplus x_2 \oplus \ldots \oplus x_n$.) Each of these `STM` actions itself returns an `IO` action, namely *the thing to be done when the choice is made.* That is why each action in the list has the cool type `STM (IO ())`. The code for `choose` first makes an atomic choice among the list of alternatives, getting the action `act`, with type `IO ()` in return; and then performs the action `act`. The list `actions` is defined in the `where` clause by taking each pair `(guard,rhs)` from the list `choices`, running the guard (an `STM` action), and returning the `IO` action gotten by applying the `rhs` to the `guard`'s return value.

4.5 Compiling and running the program

I have presented *all* the code for this example. If you simply add the appropriate import statements at the top, you should be good to go[16]:

```
module Main where
  import Control.Concurrent.STM
  import Control.Concurrent
  import System.Random
```

To compile the code, use the Glasgow Haskell Compiler, GHC[17]:

```
$ ghc Santa.hs -package stm -o santa
```

Finally you can run the program:

```
$ ./santa
----------
```

[16]You can get the code online at `http://research.microsoft.com/~simonpj/papers/stm/Santa.hs.gz`

[17]GHC is available for free at `http://haskell.org/ghc`

```
Ho! Ho! Ho! let's deliver toys
Reindeer 8 delivering toys
Reindeer 7 delivering toys
Reindeer 6 delivering toys
Reindeer 5 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 2 delivering toys
Reindeer 1 delivering toys
Reindeer 9 delivering toys
----------
Ho! Ho! Ho! let's meet in my study
Elf 3 meeting in the study
Elf 2 meeting in the study
Elf 1 meeting in the study
...and so on...
```

5 Reflections on Haskell

Haskell is, first and foremost, a *functional* language. Nevertheless, I think that it is also the world's most beautiful *imperative* language. Considered as an imperative language, Haskell's unusual features are that

- Actions (which have effects) are rigorously distinguished from pure values by the type system.
- Actions are first-class values. They can be passed to functions, returned as results, formed into lists, and so on, all without causing any side effects.

Using actions as first-class values, the programmer can define *application-specific control structures*, rather than make do with the ones provided by the language designer. For example, `nTimes` is a simple `for` loop, and `choose` implements a sort of guarded command. We also saw other applications of actions as values. In the main program we used Haskell's rich expression language (in this case list comprehensions) to generate a list of actions, which we then performed in order, using `sequence_`. Earlier, when defining `helper1`, we improved modularity by abstracting out an action from a chunk of code. To illustrate these points I have perhaps over-used Haskell's abstraction power in the Santa code, which is a very small program. For large programs, though, it is hard to overstate the importance of actions as values.

On the other hand, I have under-played other aspects of Haskell — higher order functions, lazy evaluation, data types, polymorphism, type classes, and so on — because of the focus on concurrency. Not many Haskell programs are as imperative as this one! You can find a great deal of information about

Haskell at `http://haskell.org`, including books, tutorials, Haskell compilers and interpreters, Haskell libraries, mailing lists and so on.

6 Conclusion

My main goal is to persuade you that you can write programs in a fundamentally more modular way using STM than you can with locks and condition variables. First, though, note that transactional memory allows us to completely avoid many of the standard problems that plague lock-based concurrent programs (Section 2.2). *None of these problems arise in STM Haskell.* The type system prevents you reading or writing a `TVar` outside an atomic block, and since there *are* no programmer-visible locks, the questions of which locks to take, and in which order, simply do not arise. Other benefits of STM, which I lack the space to describe here, include freedom from lost wake-ups and the treatment of exceptions and error recovery.

However, as we also discussed in Section 2.2, the worst problem with lock-based programming is that *locks do not compose.* In contrast, any function with an `STM` type in Haskell can be composed, using sequencing or choice, with any other function with an `STM` type to make a new function of `STM` type. Furthermore, the compound function will guarantee all the same atomicity properties that the individual functions did. In particular, blocking (`retry`) and choice (`orElse`), which are fundamentally non-modular when expressed using locks, are fully modular in STM. For example, consider this transaction, which uses functions we defined in Section 3.4.

```
atomically (do { limitedWithdraw a1 10
               ; limitedWithdraw2 a2 a3 20 })
```

This transaction blocks until `a1` contains at least 10 units, and either `a2` or `a3` has 20 units. However, that complicated blocking condition is not written explicitly by the programmer, and indeed if the `limitedWithdraw` functions are implemented in a sophisticated library the programmer might have no idea what their blocking conditions are. STM is modular: small programs can be glued together to make larger programs *without exposing their implementations.*

There are many aspects of transactional memory that I have not covered in this brief overview, including important topics such as nested transactions, exceptions, progress, starvation, and invariants. You can find many of them discussed in papers about STM Haskell [4, 5, 3].

Transactional memory is a particularly good "fit" for Haskell. In STM, the implementation potentially must track every memory load and store, but a Haskell STM need only track `TVar` operations, and these form only a tiny fraction of all the memory loads and stores executed by a Haskell program. Furthermore, the treatment of actions as first-class values, and the rich type system, allow us to offer strong static guarantees without extending the language in any way.

However, there is nothing to stop the adoption of transactional memory in mainstream imperative languages, although it may be less elegant and require more language support. Indeed doing so is a hot research topic: Larus and Rajwar give a comprehensive summary [6].

Using STM is like using a high-level language instead of assembly code – you can still write buggy programs, but many tricky bugs simply cannot occur, and it is much easier to focus attention on the higher-level aspects of the program. There is, alas, no silver bullet that will make concurrent programs easy to write. But STM looks like a promising step forward, and one that will help you write beautiful code.

Acknowledgements

I would like to thank those who helped me to improve the chapter with their feedback: Bo Adler, Justin Bailey, Matthew Brecknell, Paul Brown, Conal Elliot, Tony Finch, Kathleen Fisher, Greg Fitzgerald, Benjamin Franksen, Jeremy Gibbons, Tim Harris, Robert Helgesson, Dean Herington, David House, Brian Hulley, Dale Jordan, Marnix Klooster, Chris Kuklewicz, Evan Martin, Greg Meredith, Neil Mitchell, Jun Mukai, Michal Palka, Zhang Ruochen, Sebastian Sylvan, Johan Tibell, Aruthur van Leeuwen, Wim Vanderbauwhede, David Wakeling, Dan Wang, Eric Willigers Peter Wasilko, Gaal Yahas, and Brian Zimmer. My special thanks go to Kirsten Chevalier, Andy Oram, and Greg Wilson, for their particularly detailed reviews.

References

[1] Mordechai Ben-Ari. How to solve the Santa Claus problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.

[2] Nick Benton. Jingle bells: Solving the Santa Claus problem in Polyphonic C#. Technical report, Microsoft Research, 2003.

[3] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock-free data structures using STMs in Haskell. In *Eighth International Symposium on Functional and Logic Programming (FLOPS'06)*, April 2006.

[4] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, June 2005.

[5] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers,*

and Hardware Support for Transactional Computing (TRANSACT'06), Ottowa, June 2006. ACM.

[6] James Larus and Ravi Rajwar. *Transactional memory.* Morgan & Claypool, 2006.

[7] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[8] J. K. Ousterhout. Why threads are a bad idea (for most purposes), January 1996. Invited Talk, USENIX Technical Conference.

[9] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In CAR Hoare, M Broy, and R Steinbrueggen, editors, *Engineering Theories of Software Construction, Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press, 2001.

[10] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In J Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[11] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005.

[12] Herb Sutter and James Larus. Sofware and the concurrency revolution. *ACM Queue*, 3, September 2005.

[13] SJ Thompson. *Haskell: the craft of functional programming.* Addison Wesley, 1999.

[14] JA Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26:8–10, 1994.

[15] PL Wadler. The essence of functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14. ACM, Albuquerque, January 1992.

[16] PL Wadler and S Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas.* ACM, January 1989.