

# Beautiful concurrency

Simon Peyton Jones, Microsoft Research, Cambridge

December 22, 2006

## 1 Introduction

Intel tells us that the free lunch is over [7]. We can no longer expect our programs to go faster when we buy a next-generation processor. 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 program to run faster, we must learn to write parallel programs [8].

Parallel programs execute in a non-deterministic way, so they are hard to test, and bugs can be almost impossible to reproduce. If we want to write parallel program that work reliably, we must pay particular attention to beauty. Sadly, parallel program are often *less* beautiful than their sequential cousins; in particular they are, as well shall see, less *modular*.

In this chapter I'll describe *Software Transactional Memory* (STM), a promising new approach to programming shared-memory parallel processors. Although still in its infancy, STM seems to support modular programs in a way that current technology does not. 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.

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 Software Transactional Memory

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 interaction with databases is required. However, 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.

This example is somewhat unrealistic, but its very simplicity allows us to focus on what is new: the language Haskell (Section 2.1), and transactional memory (Sections 2.2 onwards). Furthermore, although it is simple, the example exposes many of the shortcomings of current technology for concurrency (Section 5).

## 2.1 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 first line gives the *type signature* for `transfer`. This signature says that `transfer` takes as its arguments 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 “when called, `transfer` may have some side effects, and then returns a value of type `()`”. The type `()`, pronounced “unit”, has just one value, which is also written `()`; so `transfer`’s result type announces that its side effects constitute the only reason for calling it. Before we can go further, we explain how side effects are handled in Haskell.

A “side effect” is anything that reads or writes mutable state that is external to `transfer`. Input/output is a prominent example of a side effect. For example, here are some 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")`<sup>1</sup> is an action that, when performed, will print “hello” on handle `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 the string thus read. We can glue together little side-effecting programs to make bigger side-effecting programs using Haskell’s “do” notation:

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

---

<sup>1</sup>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")`.

The notation `do {a1; ...; an}` constructs an action by gluing together the smaller actions `a1 ... an` 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 by “I just read:”; the `(++)` operator concatenates two strings. 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
```

That is, `return v` is an action that, when performed, does no side effects and returns `v`. This function works on values of any type, which we indicate by using a type variable `a` in its type.

Input/output is one very 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 (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 to, or reference to, a mutable location containing a value of type `t`, a bit like `*t` in C. In the case of `incRef`, the argument has type `IORef Int` because `incRef` only applies to locations that contain an `Int`.

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! And 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. In effect, Haskell gently encourages you to write programs that make sparing use of side effects.

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

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

From looking at their types alone we can see that `f` is a pure function, with 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 effects explicit will prove very useful in the rest of this chapter.

Lastly, actions are first-class values, and may be passed as arguments as well as returned as results. For example, here is a (simplified) `for` loop, 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 simply performs the action `n` times.

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*” [6].

## 2.2 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 the action atomically. More precisely:

**Atomicity:** the effects of `atomically act` become visible to another thread all at once. This guarantees 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 `a` does not “see” any effects due to other threads. It is as if `act` takes a snapshot of the state of the world when it began running, and then executes in isolation.

A simple execution model for `atomically` is this. Suppose there is a single, global lock. Then `atomically act` grabs the lock, performs the action `at`, 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. The first is that it does not actually ensure atomicity at all: There is nothing to stop a thread writing mutable cells without wrapping these operations in a call to `atomically`, thereby destroying the isolation guarantee. Second, performance would be dreadful, because every atomic block would be serialised even if no actual interference was possible.

I will discuss the second problem shortly, in Section 2.3. Meanwhile, the first objection is easily dealt with, by using the type system. We give `atomically` the following interesting 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, much smaller. The main thing you can do in an `STM` action is to read or write a transactional variable, of type `TVar a`, very much as we could read or write `IORefs` in an `IO` action<sup>2</sup>.

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

Now we cannot forget to wrap an `atomically` around an access to a `TVar` because the type system will reject any attempt to mix `STM` and `IO` actions.

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

```
type Account = TVar Int

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

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

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, in the end, `transfer` performs four primitive read/write actions: a read and then write on account `to`, followed by a read and then write on account

---

<sup>2</sup>The nomenclature is inconsistent here; it would be more consistent to use either `TVar` and `IOVar`, or `TRef` and `IORef`. But it's too late now! For better or worse we have `TVar` and `IORef`.

from. These four actions execute atomically, and that meets the specification given at the start of Section 2.

## 2.3 Implementing transactional memory

The guarantees of atomicity and isolation that I described earlier should be all that a programmer needs to use STM. However, 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 which can be implemented in a variety of ways, some simple and some sophisticated.

One particularly attractive implementation is well established in the database world: 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 it is not found, the value is read from the `TVar` itself, and the `TVar` and value read are recorded in the log. During all of this other threads might be running their own atomic blocks, reading and writing `TVars` 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 `TVars`.

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 seen 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 quite safe to run it again. However, notice that it is crucially important that `act` contains no effects *other than* reads and writes on `TVars`. 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 see an inconsistent view of memory if other threads are modifying `xv` and `yv`. If that happens, it would be a mistake to launch the missiles, and only then discover that validation fails, and the transaction should be re-run. Fortunately, the type system neatly 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.

## 2.4 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 withdraw money from an account that would leave the account overdrawn. 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 ()
```

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

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

The semantics of `retry` is 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, because 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.

But what if 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 *choice*: the ability to choose an alternative action if the first one retries. To support choice, `STM Haskell` has one further primitive action, `orElse`, whose type is

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

```

atomically :: STM a -> IO a

retry      :: STM ()
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

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:

```

withdraw2 :: Account -> Account -> Int -> STM ()
-- (withdraw acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither have enough, it retries.
withdraw acc1 acc2 amt = orElse (withdraw acc1 amt)
                             (withdraw acc2 amt)

```

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

## 2.5 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.

## 3 The Santa Claus problem

I want to show you a complete, runnable concurrent program using STM, and a well-known example is the so-called Santa Claus problem<sup>3</sup>, originally due to Trono [9]:

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,

---

<sup>3</sup>My choice was influenced by the fact that I am writing these words on 22 December.

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].

### 3.1 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 through their task.

Rendering this informal description into Haskell gives the following code for an elf:

```
elf1 :: Group -> Int -> IO ()
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group
                        ; useGate in_gate
                        ; meetInStudy ("Elf " ++ show elf_id)
                        ; useGate out_gate }
```

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

```
meetInStudy :: String -> IO ()
meetInStudy s = putStr (s ++ " meeting in the study\n")
```

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

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

The code for reindeer is identical, except that reindeer deliver toys rather than meeting in the study. 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
                             ; useGate in_gate
                             ; do_task
                             ; useGate out_gate }

```

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

```

elf1      gp id = helper1 gp (meetInStudy ("Elf "      ++ show id))
reindeer1 gp id = helper1 gp (deliverToys ("Reindeer " ++ show id))

```

I have given all of these functions a suffix “1” because they only deal with one iteration of the helper, whereas in reality the helpers re-join the fun when they are done with their task (Section 3.3).

## 3.2 Gates and Groups

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

```

newGate      :: Int -> STM Gate
useGate      :: Gate -> IO ()
operateGate :: Gate -> IO ()

```

A `Gate` has a specified capacity, which we specify when we make a new `Gate`. A `Gate` is created closed, but Santa can open it with `operateGate`, at which point an elf can call `useGate` to go through it. Before the `Gate` is open, `useGate` blocks.

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

```

data Gate = MkGate Int (TVar Int)

newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

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

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 *constructor* `MkGate`. 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 new `Gate` by allocating a TVar, and building a `Gate` value by calling the `MkGate` constructor. Dually, `useGate` uses pattern-matching to take apart the `MkGate` constructor; then it decrements the contents of the TVar, checking that there is still capacity in the gate, very much as we did with `withdraw` (Section 2.4). Finally, `operateGate` first opens the `Gate` by writing its full capacity into the TVar; and then waits to 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 with a specified capacity, and is created empty. 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 `Gates`, so that another group of eager elves can start assembling.

Here is a possible implementation:

```
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, we define `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 `Gates`. Creating a new `Group` is simply a matter of creating new `Gates`, 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?

In many cases, it is mere convenience. For example, I could have given `newGroup` an `STM` type, simply by omitting the `atomically` in its definition; but then at each call site I would have had to write `atomically (newGroup n)` rather than merely `newGroup n`. However, this choice means that I can never call `newGroup` from within an `STM` action, even though there is no reason in principle to disallow that. For example, 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 to make bigger `STM` actions, and thereby executed atomically; but as soon as you wrap a block in `atomically`, making it an `IO` type, it can no longer be combined atomically with other actions. Or, to put it another way, a library client can 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 makes 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. Again, meaningful information is conveyed by the types; in this case, `operateGates`'s type betrays the fact that it might block.

### 3.3 The main program

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

```

main = do { elf_gp <- newGroup 3
; sequence [ elf elf_gp n | n <- [1..10]]

; rein_gp <- newGroup 9
; sequence [ reindeer gp n | n <- [1..9]]

; forever (santa elf_group rein_group) }

```

The first line creates a `Group` with capacity 3 for the elves. The second line is more mysterious; it uses a *list comprehension* to create a list of `IO` actions, and

calls `sequence` to execute them in sequence. The form `[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_gp 1, elf elf_gp 2, ..., elf elf_gp 10]
```

Each of these calls yields an IO action which spawns an elf thread, and the sequencing function just runs each of the actions in sequence:

```
sequence :: [IO a] -> IO [a]
```

An `elf` is, of course, 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:

```
elf :: Group -> Int -> IO ()
elf gp id = forkIO (forever (elf1 gp id))
```

Working inside-out, the expression `(elf1 elf_gp n)` is an IO action, and we want to repeat that action indefinitely, perhaps with a random delay in the loop. We define `forever` to do just this:

```
forever :: IO () -> IO ()
-- Repeatedly perform the action, taking a rest each time
forever act = do { act
                  ; waitTime <- getStdRandom (randomR (1, 1000000))
                  ; threadDelay waitTime
                  ; forever act }
```

So `forever` takes an IO action as its argument. First, it performs `act`; then it goes to sleep for a randomly chosen time, using `threadDelay`; and the final (tail)-recursive call closes the loop to make it repeat the same thing indefinitely. Finally, `forkIO` (which is built into STM Haskell) takes an IO action (the infinite elf action in this case), and spawns it as a separate thread.

We can use the same technique to make 9 reindeer, and we can re-use `forever` to make Santa run in a loop too. All that remains is to implement Santa himself.

### 3.4 Implementing Santa

Santa is the most interesting participant of this little drama, because he makes choices. He must wait until *either* there is a group of reindeer waiting, *or* a group of elves. Here is his code:

```
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 what (in_gate, out_gate)
  = do { putStr ("Ho! Ho! Ho! let's " ++ what ++ "\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:

```
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 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 `Gates` 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  $\oplus$  [x1, ..., xn]` returns  $x_1 \oplus x_2 \oplus \dots \oplus x_n$ .) 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 ())`. Think about it.

### 3.5 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<sup>4</sup>:

```

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

```

---

<sup>4</sup>You can get the code online at <http://research.microsoft.com/~simonpj/papers/stm/Santa.hs>

To compile the code, use the Glasgow Haskell Compiler, GHC<sup>5</sup>:

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

Finally you can run the program:

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

## 4 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.

The ability to use an action as a first-class value allows the programmer to define *application-specific control structures*, rather than make do with the ones provided by the language designer. For example, `forever` is an infinite loop with a built-in delay; 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`; and when

---

<sup>5</sup>GHC is available for free at <http://haskell.org/ghc>

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, but they have a huge impact in large program. 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.

## 5 Reflections on transactional memory

The dominant technology for coordinating concurrent programs today is *locks* and *condition variables*. In an object oriented language a lock often comes implicitly with every object, and the locking is done by *synchronised methods*, but the idea is the same.

To conclude this chapter I want to persuade you that STM allows you to write programs in a fundamentally more modular way than locks and condition variables. First, though, it is worth noting how transactional memory allows us to completely avoid many of the standard problems that plague lock-based concurrent programs. Here are some examples:

**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, so it is 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. For example, if thread A transfers money from account P to Q, while thread B simultaneously transfers money from Q to P, there is a danger that A will lock P, B will lock Q, and then each will try to lock the other account and thereby deadlock. Avoiding this deadlock is always tiresome and error-prone, and sometimes extremely difficult.

None of these problems arise in STM Haskell. Haskell’s 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. STM has other benefits too that would need more space to describe, such as freedom from lost wake-ups, or the treatment of exceptions and error recovery.

However, locks and condition variables suffer from an even worse problem: they *do not compose*. That is, they 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, suppose we have a correct implementation of `withdraw` and `deposit`, using a lock or synchronised method. We cannot use those implementations unchanged to implement `transfer`; instead we must expose the locking protocol. And once `transfer` is implemented we cannot use it in a larger context (say, to transfer money from A to D, or from B to D, depending on whether A is rich enough) without exposing its locking strategy.

In contrast, any function with an STM type in Haskell can be composed with any other function with an STM type, using sequencing or choice, to make a new function of STM type; and the compound function will guarantee all the same atomicity properties that the individual functions did.

## 6 Conclusion

There are many aspects of transactional memory that I have not covered in this brief overview, including important topics such as exceptions, progress, starvation, and invariants. You can find many of them discussed in papers about STM Haskell [4, 5, 3]. However, although I have focused on STM in Haskell because it fits in so elegantly, there is nothing to stop the adoption of transactional memory in mainstream imperative languages. Indeed doing so is a hot research topic [?].

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, and beautiful, step forward.

## 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. 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. June 2006.
- [6] 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.
- [7] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, March 2005.
- [8] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3, September 2005.
- [9] JA Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26:8–10, 1994.