Parallel and Concurrent Programming in Haskell

version cadarache

Simon Marlow
simonmar@microsoft.com
Microsoft Research Ltd., Cambridge, U.K.

May 18, 2012

Contents

1 Introduction

While most programming languages nowadays provide some form of concurrent or parallel programming facilities, very few provide as wide a range as Haskell. The Haskell language is fertile ground on which to build abstractions, and concurrency and parallelism are no exception here. In the world of concurrency and parallelism, there is good reason to believe that no *one size fits all* programming model for concurrency and parallelism exists, and so prematurely committing to one particular paradigm is likely to tilt the language towards favouring certain kinds of problem. Hence in Haskell we focus on providing a wide range of abstractions and libraries, so that for any given problem it should be possible to find a tool that suits the task at hand.

In this tutorial I will introduce the main programming models available for concurrent and parallel programming in Haskell. The tutorial is woefully incomplete — there is simply too much ground to cover, but it is my hope that future revisions of this document will expand its coverage. In the meantime it should serve as an introduction to the fundamental concepts

through the use of practical examples, together with pointers to further reading for those who wish to find out more.

This tutorial takes a deliberately practical approach: most of the examples are real Haskell programs that you can compile, run, measure, modify and experiment with. For information on how to obtain the code samples, see Section 2.1. There is also a set of accompanying exercises.

In order to follow this tutorial you should have a basic knowledge of Haskell, including programming with monads.

Briefly, the topics covered in this tutorial are as follows:

- Parallel programming with the `Eval` monad (Section 4)
- Evaluation Strategies (Section 5)
- Dataflow parallelism with the `Par` monad (Section 6)
- GPU programming with the *Accelerate* library (Section 7)
- Basic Concurrent Haskell (Part II)
- Asynchronous exceptions (Section 10)
- Software Transactional Memory (Section 11)
- Concurrency and the Foreign Function Interface (Section 15)
- High-speed concurrent servers (Section 14)
- Distributed programming (Section 17)

One useful aspect of this tutorial as compared to previous tutorials covering similar ground ([12; 13]) is that I have been able to take into account recent changes to the APIs. In particular, the `Eval` monad has replaced `par` and `pseq` (thankfully), and in asynchronous exceptions `mask` has replaced the old `block` and `unblock`.

2 Tools and resources

To try out Parallel and Concurrent Haskell, and to run the sample programs that accompany this article, you will need to install the Haskell Platform[1]. The Haskell Platform includes the GHC compiler and all the important libraries, including the parallel and concurrent libraries we shall be using. This version of the tutorial was tested with the Haskell Platform version 2011.2.0.1, and we expect to update this tutorial as necessary to cover future changes in the platform.

Section 6 requires the `monad-par` package, which is not currently part of the Haskell Platform. To install it, use the `cabal` command:

[1] `http://hackage.haskell.org/platform/`

```
$ cabal install monad-par
```

(The examples in this tutorial were tested with `monad-par` version 0.1.0.3).

Additionally, we recommend installing ThreadScope[2]. ThreadScope is a tool for visualising the execution of Haskell programs, and is particularly useful for gaining insight into the behaviour of parallel and concurrent Haskell code. On some systems (mainly Linux) ThreadScope can be installed with a simple

```
$ cabal install threadscope
```

but for other systems refer to the ThreadScope documentation at the aforementioned URL.

While reading the article we recommend you have the following documentation to hand:

- The GHC User's Guide[3],
- The Haskell Platform library documentation, which can be found on the main Haskell Platform site[4]. Any types or functions that we use in this article that are not explicitly described can be found documented there.

It should be noted that none of the APIs described in this tutorial are *standard* in the sense of being part of the Haskell specification. That may change in the future.

2.1 Sample Code

The repository containing the source for both this document and the code samples can be found at `https://github.com/simonmar/par-tutorial`. The current version can be downloaded from `http://community.haskell.org/~simonmar/par-tutorial-cadarache.zip`.

3 Terminology: Parallelism and Concurrency

In many fields, the words *parallel* and *concurrent* are synonyms; not so in programming, where they are used to describe fundamentally different concepts.

A *parallel* program is one that uses a multiplicity of computational hardware (e.g. multiple processor cores) in order to perform computation more

[2] `http://www.haskell.org/haskellwiki/ThreadScope`

[3] `http://www.haskell.org/ghc/docs/latest/html/users_guide/`

[4] `http://hackage.haskell.org/platform/`

quickly. Different parts of the computation are delegated to different processors that execute at the same time (in *parallel*), so that results may be delivered earlier than if the computation had been performed sequentially.

In contrast, *concurrency* is a program-structuring technique in which there are multiple *threads of control*. Notionally the threads of control execute "at the same time"; that is, the user sees their effects interleaved. Whether they actually execute at the same time or not is an implementation detail; a concurrent program can execute on a single processor through interleaved execution, or on multiple physical processors.

While parallel programming is concerned only with efficiency, concurrent programming is concerned with structuring a program that needs to interact with multiple independent external agents (for example the user, a database server, and some external clients). Concurrency allows such programs to be *modular*; the thread that interacts with the user is distinct from the thread that talks to the database. In the absence of concurrency, such programs have to be written with event loops and callbacks—indeed, event loops and callbacks are often used even when concurrency is available, because in many languages concurrency is either too expensive, or too difficult, to use.

The notion of "threads of control" does not make sense in a purely functional program, because there are no effects to observe, and the evaluation order is irrelevant. So concurrency is a structuring technique for effectful code; in Haskell, that means code in the `IO` monad.

A related distinction is between *deterministic* and *nondeterministic* programming models. A deterministic programming model is one in which each program can give only one result, whereas a nondeterministic programming model admits programs that may have different results, depending on some aspect of the execution. Concurrent programming models are necessarily nondeterministic, because they must interact with external agents that cause events at unpredictable times. Nondeterminism has some notable drawbacks, however: programs become significantly harder to test and reason about.

For parallel programming we would like to use deterministic programming models if at all possible. Since the goal is just to arrive at the answer more quickly, we would rather not make our program harder to debug in the process. Deterministic parallel programming is the best of both worlds: testing, debugging and reasoning can be performed on the sequential program, but the program runs faster when processors are added. Indeed, most computer processors themselves implement deterministic parallelism in the form of pipelining and multiple execution units.

While it is possible to do parallel programming using concurrency, that is often a poor choice, because concurrency sacrifices determinism. In Haskell, the parallel programming models are deterministic. However, it is important to note that deterministic programming models are not sufficient to express all kinds of parallel algorithms; there are algorithms that depend

on internal nondeterminism, particularly problems that involve searching a solution space. In Haskell, this class of algorithms is expressible only using concurrency.

Finally, it is entirely reasonable to want to mix parallelism and concurrency in the same program. Most interactive programs will need to use concurrency to maintain a responsive user interface while the compute intensive tasks are being performed.

Part I
Parallel Haskell

Parallel Haskell is all about making Haskell programs run *faster* by dividing the work to be done between multiple processors. Now that processor manufacturers have largely given up trying to squeeze more performance out of individual processors and have refocussed their attention on providing us with more processors instead, the biggest gains in performance are to be had by using parallel techniques in our programs so as to make use of these extra cores.

We might wonder whether the compiler could automatically parallelise programs for us. After all, it should be easier to do this in a pure functional language where the only dependencies between computations are data dependencies, and those are mostly perspicuous and thus readily analysed. In contrast, when effects are unrestricted, analysis of dependencies tends to be much harder, leading to greater approximation and a large degree of false dependencies. However, even in a language with only data dependencies, automatic parallelisation still suffers from an age-old problem: managing parallel tasks requires some bookkeeping relative to sequential execution and thus has an inherent overhead, so the size of the parallel tasks must be large enough to overcome the overhead. Analysing costs at compile time is hard, so one approach is to use runtime profiling to find tasks that are costly enough and can also be run in parallel, and feed this information back into the compiler. Even this, however, has not been terribly successful in practice [1].

Fully automatic parallelisation is still a pipe dream. However, the parallel programming models provided by Haskell do succeed in eliminating some mundane or error-prone aspects traditionally associated with parallel programming:

- Parallel programming in Haskell is *deterministic*: the parallel program always produces the same answer, regardless how many processors are used to run it, so parallel programs can be debugged without actually running them in parallel.

- Parallel Haskell programs do not explicitly deal with *synchronisation* or *communication*. Synchronisation is the act of waiting for other tasks to complete, perhaps due to data dependencies. Communication involves the transmission of results between tasks running on different processors. Synchronisation is handled automatically by the GHC runtime system and/or the parallelism libraries. Communication is implicit in GHC since all tasks share the same heap, and can share objects without restriction. In this setting, although there is no explicit communication at the program level or even the runtime level, at the hardware level communication re-emerges as the transmission of data between the caches of the different cores. Excessive communication can cause contention for the main memory bus, and such overheads can be difficult to diagnose.

Parallel Haskell does require the programmer to think about **Partitioning**. The programmer's job is to subdivide the work into tasks that can execute in parallel. Ideally, we want to have enough tasks that we can keep all the processors busy for the entire runtime. However, our efforts may be thwarted:

- **Granularity**. If we make our tasks too small, then the overhead of managing the tasks outweighs any benefit we might get from running them in parallel. So granularity should be large enough to dwarf the overheads, but not too large, because then we risk not having enough work to keep all the processors busy, especially towards the end of the execution when there are fewer tasks left.

- **Data dependencies** between tasks enforce sequentialisation. GHC's two parallel programming models take different approaches to data dependencies: in *Strategies* (Section 5), data dependencies are entirely implicit, whereas in the *Par monad* (Section 6), they are explicit. This makes programming with Strategies somewhat more concise, at the expense of the possibility that hidden dependencies could cause sequentialisation at runtime.

In this tutorial we will describe two parallel programming models provided by GHC. The first, *Evaluation Strategies* [8] (Strategies for short), is well-established and there are many good examples of using Strategies to write parallel Haskell programs. The second is a dataflow programming model based around a `Par` monad [5]. This is a newer programming model in which it is possible to express parallel coordination more explicitly than with Strategies, though at the expense of some of the conciseness and modularity of Strategies.

4 Basic parallelism: the `Eval` monad

In this section we will demonstrate how to use the basic parallelism abstractions in Haskell to perform some computations in parallel. As a running example that you can actually test yourself, we use a Sudoku solver[5]. The Sudoku solver is very fast, and can solve all 49,000 of the known puzzles with 17 clues[6] in about 2 minutes.

We start with some ordinary sequential code to solve a set of Sudoku problems read from a file:

```
import Sudoku
import Control.Exception
import System.Environment

main :: IO ()
main = do
    [f] <- getArgs
    grids <- fmap lines $ readFile f
    mapM_ (evaluate . solve) grids
```

The module `Sudoku` provides us with a function `solve` with type

```
solve :: String -> Maybe Grid
```

where the `String` represents a single Sudoku problem, and `Grid` is a representation of the solution. The function returns `Nothing` if the problem has no solution. For the purposes of this example we are not interested in the solution itself, so our `main` function simply calls `evaluate . solve` on each line of the file (the file will contain one Sudoku problem per line). The `evaluate` function comes from `Control.Exception` and has type

```
evaluate :: a -> IO a
```

It evaluates its argument to *weak-head normal form*. Weak-head normal form just means that the expression is evaluated as far as the first constructor; for example, if the expression is a list, then `evaluate` would perform enough evaluation to determine whether the list is empty (`[]`) or non-empty (`_:_`), but it would not evaluate the head or tail of the list. The `evaluate` function returns its result in the `IO` monad, so it is useful for forcing evaluation at a particular time.

Compile the program as follows:

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
```

and run it on 1000 sample problems:

[5] The Sudoku solver code can be found in the module `Sudoku.hs` in the samples that accompany this tutorial.

[6] `http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php`

```
$ ./sudoku1 sudoku17.1000.txt +RTS -s
./sudoku1 sudoku17.1000.txt +RTS -s
   2,392,127,440 bytes allocated in the heap
      36,829,592 bytes copied during GC
         191,168 bytes maximum residency (11 sample(s))
          82,256 bytes maximum slop
               2 MB total memory in use

  Generation 0:  4570 collections, 0 parallel, 0.14s, 0.13s elapsed
  Generation 1:    11 collections, 0 parallel, 0.00s, 0.00s elapsed

  Parallel GC work balance: -nan (0 / 0, ideal 1)

                        MUT time (elapsed)       GC time  (elapsed)
  Task  0 (worker) :    0.00s    (  0.00s)       0.00s    (  0.00s)
  Task  1 (worker) :    0.00s    (  2.92s)       0.00s    (  0.00s)
  Task  2 (bound)  :    2.92s    (  2.92s)       0.14s    (  0.14s)

  SPARKS: 0 (0 converted, 0 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.92s  (  2.92s elapsed)
  GC    time    0.14s  (  0.14s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.06s  (  3.06s elapsed)

  %GC time       4.6%  (4.6% elapsed)

  Alloc rate    818,892,766 bytes per MUT second

  Productivity  95.4% of total user, 95.3% of total elapsed
```

The argument `+RTS -s` instructs the GHC runtime system to emit the statistics you see above. These are particularly helpful as a first step in analysing parallel performance. The output is explained in detail in the GHC User's Guide, but for our purposes we are interested in one particular metric: `Total time`. This figure is given in two forms: the first is the total CPU time used by the program, and the second figure is the *elapsed*, or wall-clock, time. Since we are running on a single processor, these times are identical (sometimes the elapsed time might be slightly larger due to other activity on the system).

This program should parallelise quite easily; after all, each problem can be solved completely independently of the others. First, we will need some basic functionality for expressing parallelism, which is provided by the module `Control.Parallel.Strategies`:

```
data Eval a
instance Monad Eval
```

```
runEval :: Eval a -> a

rpar :: a -> Eval a
rseq :: a -> Eval a
```

Parallel coordination will be performed in a monad, namely the `Eval` monad. The reason for this is that parallel programming fundamentally involves *ordering* things: start evaluating `a` in parallel, *and then* evaluate `b`. Monads are good for expressing ordering relationships in a compositional way.

The `Eval` monad provides a `runEval` operation that lets us extract the value from `Eval`. Note that `runEval` is completely pure - there's no need to be in the `IO` monad here.

The `Eval` monad comes with two basic operations, `rpar` and `rseq`. The `rpar` combinator is used for creating parallelism; it says "my argument could be evaluated in parallel", while `rseq` is used for forcing sequential evaluation: it says "evaluate my argument now" (to weak-head normal form). These two operations are typicaly used together - for example, to evaluate `A` and `B` in parallel, we could apply `rpar` on `A`, followed by `rseq` on `B`.

Returning to our Sudoku example, let us add some parallelism to make use of two processors. We have a list of problems to solve, so it should suffice to divide the list in two and solve the problems in each half of the list in parallel. Here is some code to do just that[7]:

```
1    let (as,bs) = splitAt (length grids `div` 2) grids

3    evaluate $ runEval $ do
4       a <- rpar (deep (map solve as))
5       b <- rpar (deep (map solve bs))
6       rseq a
7       rseq b
8       return ()
```

line 1 divides the list into two equal (or nearly-equal) sub-lists, `as` and `bs`. The next part needs more explanation:

3 We are going to `evaluate` an application of `runEval`

4 Create a parallel task to compute the solutions to the problems in the sub-list `as`. The expression `map solve as` represents the solutions; however, just evaluating this expression to weak-head normal form will not actually compute any of the solutions, since it will only evaluate as far as the first `(:)` cell of the list. We need to fully evaluate the whole list, including the elements. This is why we added an application of the `deep` function, which is defined as follows:

```
deep :: NFData a => a -> a
deep a = deepseq a a
```

[7] full code in sample `sudoku2.hs`

`deep` evaluates the entire structure of its argument (reducing it to *normal form*), before returning the argument itself. It is defined in terms of the function `deepseq`, which is available from the `Control.DeepSeq` module.

Not evaluating deeply enough is a common mistake when using the `rpar` monad, so it is a good idea to get into the habit of thinking, for each `rpar`, "how much of this structure do I want to evaluate in the parallel task?" (indeed, it is such a common problem that in the `Par` monad to be introduced later, we went so far as to make `deepseq` the default behaviour).

5 Create a parallel task to compute the solutions to `bs`, exactly as for `as`.

6-7 Using `rseq`, we wait for both parallel tasks to complete.

8 Finally, return (for this example we aren't interested in the results themselves, only in the act of computing them).

In order to use parallelism with GHC, we have to add the `-threaded` option, like so:

```
$ ghc -O2 sudoku2.hs -rtsopts -threaded
[2 of 2] Compiling Main             ( sudoku2.hs, sudoku2.o )
Linking sudoku2 ...
```

Now, we can run the program using 2 processors:

```
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -s
./sudoku2 sudoku17.1000.txt +RTS -N2 -s
   2,400,125,664 bytes allocated in the heap
      48,845,008 bytes copied during GC
       2,617,120 bytes maximum residency (7 sample(s))
         313,496 bytes maximum slop
               9 MB total memory in use

  Gen 0: 2975 collections, 2974 parallel, 1.04s, 0.15s elapsed
  Gen 1:    7 collections,    7 parallel, 0.05s, 0.02s elapsed

  Parallel GC work balance: 1.52 (6087267 / 3999565, ideal 2)

                        MUT time (elapsed)       GC time  (elapsed)
  Task  0 (worker) :    1.27s    (  1.80s)       0.69s    (  0.10s)
  Task  1 (worker) :    0.00s    (  1.80s)       0.00s    (  0.00s)
  Task  2 (bound)  :    0.88s    (  1.80s)       0.39s    (  0.07s)
  Task  3 (worker) :    0.05s    (  1.80s)       0.00s    (  0.00s)

  SPARKS: 2 (1 converted, 0 pruned)
```

```
  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.21s  (  1.80s elapsed)
  GC    time    1.08s  (  0.17s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    3.29s  (  1.97s elapsed)

  %GC time      32.9%  (8.8% elapsed)

  Alloc rate    1,087,049,866 bytes per MUT second

  Productivity  67.0% of total user, 111.9% of total elapsed
```

Note that the `Total time` now shows a marked difference between the CPU time (3.29s) and the elapsed time (1.97s). Previously the elapsed time was 3.06s, so we can calculate the *speedup* on 2 processors as $3.06/1.97 = 1.55$. Speedups are always calculated as a ratio of wall-clock times. The CPU time is a helpful metric for telling us how busy our processors are, but as you can see here, the CPU time when running on multiple processors is often greater than the wall-clock time for a single processor, so it would be misleading to calculate the speedup as the ratio of CPU time to wall-clock time (1.67 here).

Why is the speedup only 1.55, and not 2? In general there could be a host of reasons for this, not all of which are under the control of the Haskell programmer. However, in this case the problem is partly of our doing, and we can diagnose it using the ThreadScope tool. To profile the program using ThreadScope we need to first recompile it with the `-eventlog` flag, run it with `+RTS -ls`, and then invoke ThreadScope on the generated `sudoku2.eventlog` file:

```
$ rm sudoku2; ghc -O2 sudoku2.hs -threaded -rtsopts -eventlog
[2 of 2] Compiling Main             ( sudoku2.hs, sudoku2.o )
Linking sudoku2 ...
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -ls
$ threadscope sudoku2.eventlog
```

The ThreadScope profile is shown in Figure 1; this graph was generated by selecting "export to PNG" from ThreadScope, so it includes the timeline graph only, and not the rest of the ThreadScope GUI. The x axis of the graph is time, and there are three horizontal bars showing how the program executed over time. The topmost bar is known as the "activity" profile, and it shows how many processors were executing Haskell code (as opposed to being idle or garbage collecting) at a given point in time. Underneath the activity profile there is one bar per processor, showing what that processor was doing at each point in the execution. Each bar has two parts:: the upper, thicker bar is green when that processor is executing Haskell code, and the

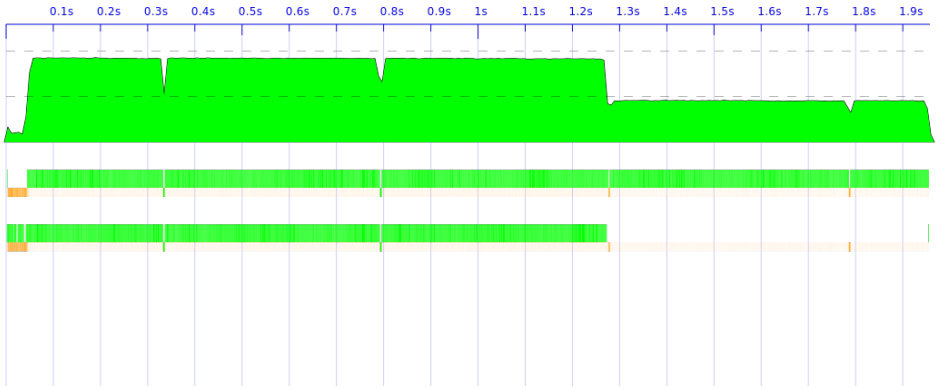


Figure 1: Sudoku2 ThreadScope profile

lower, narrower bar is orange or green when that processor is performing garbage collection.[8]

As we can see from the graph, there is a period at the end of the run where just one processor is executing, and the other one is idle (except for participating in regular garbage collections, which is necessary for GHC's parallel garbage collector). This indicates that our two parallel tasks are uneven: one takes much longer to execute than the other, and so we are not making full use of our 2 processors, which results in less than perfect speedup.

Why should the workloads be uneven? After all, we divided the list in two, and we know the sample input has an even number of problems. The reason for the unevenness is that each problem does not take the same amount of time to solve, it all depends on the searching strategy used by the Sudoku solver[9]. This illustrates an important distinction between two partitioning strategies:

- **Static Partitioning**, which is the technique we used to partition the Sudoku problems here, consists of dividing the work according to some pre-defined policy (here, dividing the list equally in two).
- **Dynamic Partitioning** instead tries to distribute the work more evenly, by dividing the work into smaller tasks and only assigning tasks to processors when they are idle.

The GHC runtime system supports automatic distribution of the parallel tasks; all we have to do to achieve dynamic partitioning is divide the problem

[8] the distinction between orange and green during GC has to do with the kind of GC activity being performed, and need not concern us here.

[9] In fact, we ordered the problems in the sample input so as to clearly demonstrate the problem.

into small enough tasks and the runtime will do the rest for us.

The argument to `rpar` is called a *spark*. The runtime collects sparks in a pool and uses this as a source of work to do when there are spare processors available, using a technique called *work stealing* [7]. Sparks may be evaluated at some point in the future, or they might not — it all depends on whether there is spare processor capacity available. Sparks are very cheap to create (`rpar` essentially just adds a reference to the expression to an array).

So, let's try using dynamic partitioning with the Sudoku problem. First we define an abstraction that will let us apply a function to a list in parallel, `parMap`:

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f [] = return []
parMap f (a:as) = do
   b <- rpar (f a)
   bs <- parMap f as
   return (b:bs)
```

This is rather like a monadic version of `map`, except that we have used `rpar` to lift the application of the function `f` to the element `a` into the `Eval` monad. Hence, `parMap` runs down the whole list, eagerly creating sparks for the application of `f` to each element, and finally returns the new list. When `parMap` returns, it will have created one spark for each element of the list.

We still need to evaluate the result list itself, and that is straightforward with `deep`:

```
    evaluate $ deep $ runEval $ parMap solve grids
```

Running this new version[10] yields more speedup:

```
Total time    3.55s  (  1.79s elapsed)
```

which we can calculate is equivalent to a speedup of $3.06/1.79 = 1.7$, approaching the ideal speedup of 2. Furthermore, the GHC runtime system tells us how many sparks were created:

```
SPARKS: 1000 (1000 converted, 0 pruned)
```

we created exactly 1000 sparks, and they were all *converted* (that is, turned into real parallelism at runtime). Sparks that are *pruned* have been removed from the spark pool by the runtime system, either because they were found to be already evaluated, or because they were found to be not referenced by the rest of the program, and so are deemed to be not useful. We will discuss the latter requirement in more detail in Section 5.1.

The ThreadScope profile looks much better (Figure 2). Furthermore, now that the runtime is managing the work distribution for us, the program will automatically scale to more processors. On an 8 processor machine, for example:

[10] code sample `sudoku3.hs`

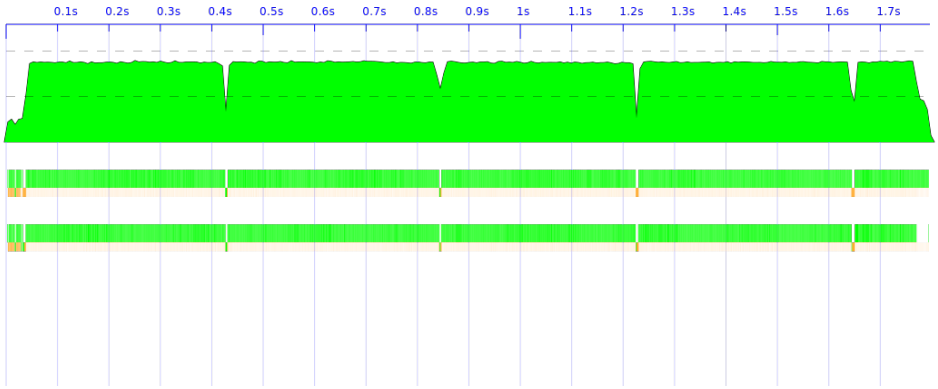


Figure 2: Sudoku3 ThreadScope profile

Figure 3: Sudoku3 (zoomed) ThreadScope profile

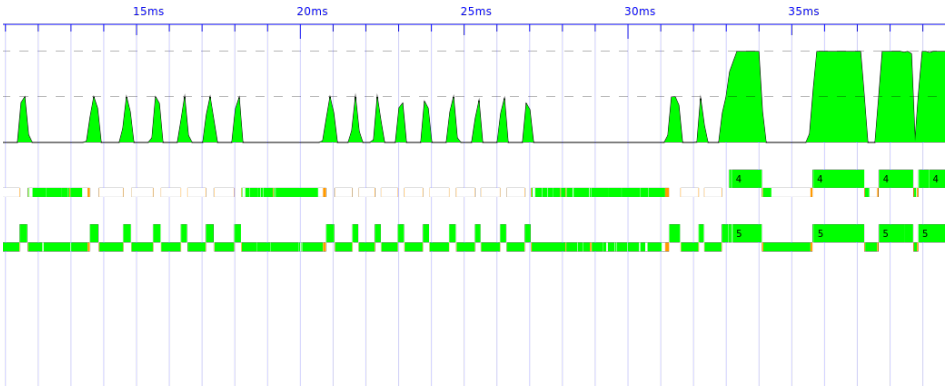


Figure 4: Sudoku4 ThreadScope profile

```
Total time    4.46s  (  0.59s elapsed)
```

which equates to a speedup of 5.2 over the sequential version.

If we look closely at the 2-processor profile there appears to be a short section near the beginning where not much work is happening. In fact, zooming in on this section in ThreadScope (Figure 3) reveals that both processors are working, but most of the activity is garbage collection, and only one processor is performing most of the garbage collection work. In fact, what we are seeing here is the program reading the input file (lazily) and dividing it into lines, driven by the demand of `parMap` which traverses the whole list of lines.

Since reading the file and dividing it into lines is a sequential activity anyway, we could force it to happen all at once before we start the main computation, by adding

```
    evaluate (length grids)
```

(see code sample `sudoku4.hs`). This makes no difference to the overall runtime, but it divides the execution into sequential and parallel parts, as we can see in ThreadScope (Figure 4).

Now, we can read off the portion of the runtime that is sequential: 33ms. When we have a sequential portion of our program, this affects the maximum parallel speedup that is achievable, which we can calculate using Amdahl's law. Amdahl's law gives the maximum achievable speedup as the ratio

$$\frac{1}{(1-P) + \frac{P}{N}}$$

where P is the portion of the runtime that can be parallelised, and N is the number of processors available. In our case, P is $(3.06 - 0.033)/3.06 = 0.9892$, and the maximum speedup is hence 1.98. The sequential fraction

here is too small to make a significant impact on the theoretical maximum speedup with 2 processors, but when we have more processors, say 64, it becomes much more important: $1/((1 - 0.989) + 0.989/64) = 38.1$. So no matter what we do, this tiny sequential part of our program will limit the maximum speedup we can obtain with 64 processors to 38.1. In fact, even with 1024 cores we could only achieve around 84 speedup, and it is impossible to achieve a speedup of 91 no matter how many cores we have. Amdahl's law tells us that not only does parallel speedup become harder to achieve the more processors we add, in practice most programs have a theoretical maximum amount of parallelism.

ToDo: Add something about Gustafson's law http://en.wikipedia.org/wiki/Gustafson%27s_Law

5 Evaluation Strategies

Evaluation Strategies [14; 8] is an abstraction layer built on top of the `Eval` monad that allows larger parallel specifications to be built in a compositional way. Furthermore Strategies allow parallel coordination to be described in a modular way, separating parallelism from the algorithm to be parallelised.

A Strategy is merely a function in the `Eval` monad that takes a value of type `a` and returns the same value:

```
type Strategy a = a -> Eval a
```

Strategies are identity functions; that is, the value returned by a `Strategy` is observably equivalent to the value it was passed. Unfortunately the library cannot statically guarantee this property for user-defined `Strategy` functions, but it holds for the `Strategy` functions and combinators provided by the `Control.Parallel.Strategies` module.

We have already seen some simple Strategies, `rpar` and `rseq`, although we can now give their types in terms of `Strategy`:

```
rseq :: Strategy a
rpar :: Strategy a
```

There are two further members of this family:

```
r0 :: Strategy a
r0 x = return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rseq (deep x)
```

`r0` is the `Strategy` that evaluates nothing, and `rdeepseq` is the `Strategy` that evaluates the entire structure of its argument, which can be defined in terms of `deep` that we saw earlier. Note that `rseq` is necessary here: replacing `rseq` with `return` would not perform the evaluation immediately, but would defer it until the value returned by `rdeepseq` is demanded (which might be never).

We have some simple ways to build Strategies, but how is a Strategy actually *used*? A `Strategy` is just a function yielding a computation in the `Eval` monad, so we could use `runEval`. For example, applying the strategy `s` to a value `x` would be simply `runEval (s x)`. This is such a common pattern that the Strategies library gives it a name, `using`:

```
using :: a -> Strategy a -> a
x `using` s = runEval (s x)
```

`using` takes a value of type `a`, a Strategy for `a`, and applies the Strategy to the value. The identity property for `Strategy` gives us that

```
x `using` s == x
```

which is a significant benefit of Strategies: every occurrence of `` `using` s`` can be deleted without affecting the semantics. Strictly speaking there are two caveats to this property. Firstly, as mentioned earlier, user-defined `Strategy` functions might not satisfy the identity property. Secondly, the expression `` x `using` s`` might be less defined than `x`, because it evaluates more structure of `x` than the context does. So deleting `` `using` s`` might have the effect of making the program terminate with a result when it would previously throw an exception or fail to terminate. Making programs more defined is generally considered to be a somewhat benign change in semantics (indeed, GHC's optimiser can also make programs more defined under certain conditions), but nevertheless it is a change in semantics.

5.1 A Strategy for evaluating a list in parallel

In Section 4 we defined a function `parMap` that would map a function over a list in parallel. We can think of `parMap` as a composition of two parts:

- The algorithm: `map`
- The parallelism: evaluating the elements of a list in parallel

and indeed with Strategies we can express it exactly this way:

```
parMap f xs = map f xs `using` parList rseq
```

The benefits of this approach are two-fold: not only does it separate the algorithm from the parallelism, but it also *reuses* `map`, rather than re-implementing a parallel version.

The `parList` function is a Strategy on lists, defined as follows:

```
parList :: Strategy a -> Strategy [a]
parList strat []     = return []
parList strat (x:xs) = do
  x'  <- rpar (x `using` strat)
  xs' <- parList strat xs
  return (x':xs')
```

(in fact, `parList` is already provided by `Control.Parallel.Strategies` so you don't have to define it yourself, but we are using its implementation here as an illustration).

The `parList` function is a *parameterised* Strategy, that is, it takes as an argument a Strategy on values of type `a`, and returns a Strategy for lists of `a`. This illustrates another important aspect of Strategies: they are compositional, in the sense that we can build larger strategies by composing smaller reusable components. Here, `parList` describes a family of Strategies on lists that evaluate the list elements in parallel.

On line 4, `parList` calls `rpar` to create a spark to evaluate the current element of the list. Note that the spark evaluates `(x `using` strat)`: that is, it applies the argument Strategy `strat` to the list element `x`.

As `parList` traverses the list sparking list elements, it remembers each value returned by `rpar` (bound to `x'`), and constructs a new list from these values. Why? After all, this seems to be a lot of trouble to go to, because it means that `parList` is no longer *tail-recursive* — the recursive call to `parList` is not the last operation in the `do` on its right-hand side, and so `parList` will require stack space linear in the length of the input list.

Couldn't we write a tail-recursive version instead? For example:

```
parList :: Strategy a -> Strategy [a]
parList strat xs = do go xs; return xs
  where go [] = return ()
        go (x:xs) = do
           rpar (x `using` strat)
           go xs
```

This typechecks, after all, and seems to call `rpar` on each list element as required.

The difference is subtle but important, and is best understood via a diagram (Figure 5). At the top of the diagram we have the input list `xs`: a linked list of cells, each of which points to a list element (`x1`, `x2`, and so forth). At the bottom of the diagram is the *spark pool*, the runtime system data structure that stores references to sparks in the heap. The other structures in the diagram are built by `parList` (the first version). Each `strat` box represents `(x `using` strat)` for an element `x` of the original list, and `xs'` is the linked list of cells in the output list. The spark pool contains pointers to each of the `strat` boxes; these are the pointers created by the `rpar` calls.

Now, the spark pool only retains references to objects that are required by the program. If the runtime finds that the spark pool contains a reference to an object that the program will never use, then the reference is dropped, and any potential parallelism it represented is lost. This behaviour is a deliberate policy; if it weren't this way, then the spark pool could retain data indefinitely, causing a space leak (details can be found in Marlow et al. [8]).

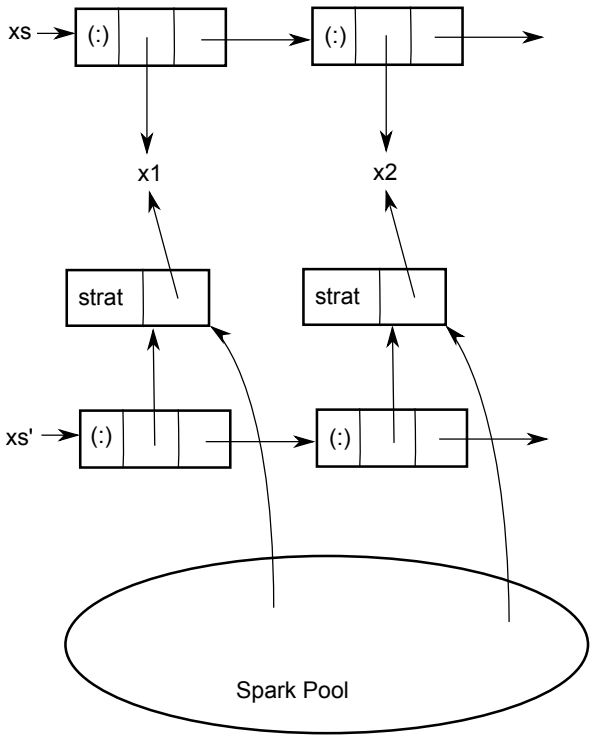


Figure 5: `parList` heap structures

This is the reason for the list `xs'`. Suppose we did not build the new list `xs'`, as in the tail-recursive version of `parList` above. Then, the only reference to each `strat` box in the heap would be from the spark pool, and hence the runtime would automatically sweep all those references from the spark pool, discarding the parallelism. Hence we build a new list `xs'`, so that the program can retain references to the sparks for as long as it needs to.

This automatic discarding of unreferenced sparks has another benefit: suppose that under some circumstances the program does not need the entire list. If the program simply forgets the unused remainder of the list, the runtime system will clean up the unreferenced sparks from the spark pool, and will not waste any further parallel processing resources on evaluating those sparks. The extra parallelism in this case is termed *speculative*, because it is not necessarily required, and the runtime will automatically discard speculative tasks that it can prove will never be required - a useful property!

While the runtime system's discarding of unreferenced sparks is certainly useful in some cases, it can be tricky to work with, because there is no language-level support for catching mistakes. Fortunately the runtime system will tell us if it garbage collects unreferenced sparks; for example:

```
SPARKS: 144 (0 converted, 144 pruned)
```

A large number of sparks being "pruned" is a good indication that sparks are being removed from the spark pool before they can be used for parallelism. Sparks can be pruned for several reasons:

- The spark was a *dud*: it was already evaluated at the point it was sparked.
- The spark *fizzled*: it was evaluated by some other thread before it could be evaluated in parallel.
- The spark was garbage collected, as described above.

In fact, GHC from version 7.2.1 onwards separates these different classifications in its output from `+RTS -s`:

```
SPARKS: 144 (0 converted, 0 dud, 144 GC'd, 0 fizzled)
```

Unless you are using speculation, then a non-zero figure for GC'd sparks is probably a bad sign.

All of the combinators in the library `Control.Parallel.Strategies` behave correctly with respect to retaining references to sparks when necessary. So the rules of thumb for not tripping up here are:

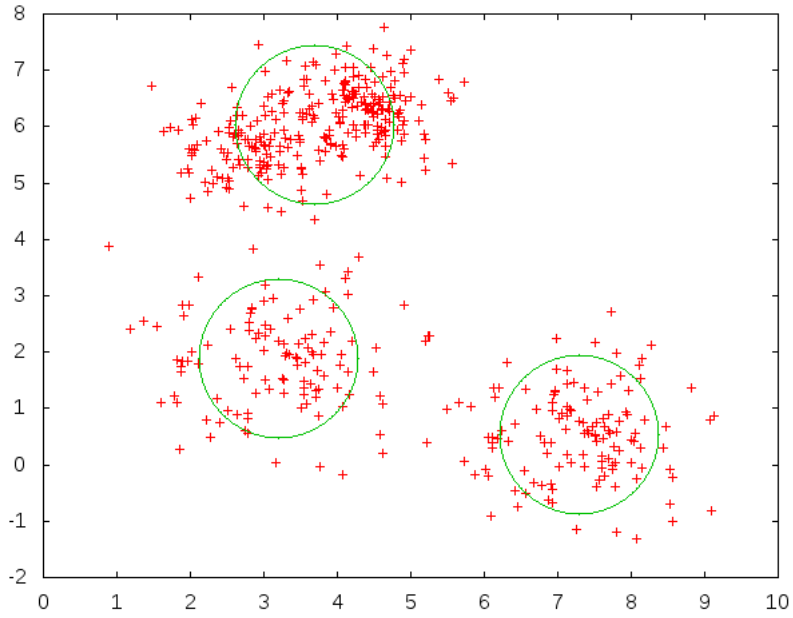


Figure 6: The K-Means problem

- Use `using` to apply strategies: it encourages the right pattern, in which the program uses the results of applying the Strategy.
- When writing your own `Eval`-monad code, remember to bind the result of `rpar`, and use its result.

5.2 Using `parList`: the K-Means problem

The `parList` Strategy covers a wide range of uses for parallelism in typical Haskell programs; in many cases, a single `parList` is all that is needed to expose sufficient parallelism.

Returning to our Sudoku solver from Section 4 for a moment, instead of our own hand-written `parMap`, we could have used `parList`:

```
evaluate $ deep $ map solve grids `using` parList rseq
```

Let's look at a slightly more involved example. In the K-Means problem, the goal is to partition a set of data points into clusters. Finding an optimal solution to the problem is NP-hard, but there exist several heuristic techniques that do not guarantee to find an optimal solution, but work well in practice. For example, given the data points shown in Figure 6, the algorithm should discover the clusters indicated by the circles. Here we have only shown the locations of the clusters, partitioning the points is achieved by simply finding the closest cluster to each point.

The most well-known heuristic technique is Lloyd's algorithm, which finds a solution by iteratively improving an initial guess, as follows:

1. Pick an initial set of clusters by randomly assigning each point in the data set to a cluster.
2. Find the centroid of each cluster (the average of all the points in the cluster).
3. Assign each point to the cluster to which it is closest, this gives a new set of clusters.
4. Repeat steps 2–3 until the set of clusters stabilises.

Of course the algorithm works in any number of dimensions, but we will use 2 for ease of visualisation.

A complete Haskell implementation can be found in the directory `kmeans` in the sample code; Figure 7 shows the core of the algorithm.

A data point is represented by the type `Vector`, which is just a pair of `Double`s. Clusters are represented by the type `Cluster`, which contains its number, the count of points assigned to this cluster, the sum of the `Vector`s in the cluster, and its centre. Everything about the cluster except its number is derivable from the set of points in the cluster; this is expressed by the function `makeCluster`. Essentially `Cluster` caches various information about a cluster, and the reason we need to cache these specific items will become clear shortly.

The function `assign` implements step 3 of the algorithm, assigning points to clusters. The `accumArray` function is particularly useful for this kind of bucket-sorting task. The function `makeNewClusters` implements step 2 of the algorithm, and finally `step` combines `assign` and `makeNewClusters` to implement one complete iteration.

To complete the algorithm we need a driver to repeatedly apply the `step` function until convergence. The function `kmeans_seq`, in Figure 8, implements this.

How can this algorithm be parallelised? One place that looks straightforward to parallelise is the `assign` function, since it is essentially just a `map` over the points. However, that doesn't get us very far: we cannot parallelise `accumArray` directly, so we would have to do multiple `accumArray`s and combine the results, and combining elements would mean an extra list append. The `makeNewClusters` operation parallelises easily, but only in so far as each `makeCluster` is independent of the others; typically the number of clusters is much smaller than the number of points (e.g. a few clusters to a few hundred thousand points), so we don't gain much scalability by parallelising `makeNewClusters`.

We would like a way to parallelise the problem at a higher level. That is, we would like to divide the set of points into chunks, and process each

```haskell
data Vector = Vector Double Double

addVector :: Vector -> Vector -> Vector
addVector (Vector a b) (Vector c d) = Vector (a+c) (b+d)

data Cluster = Cluster
               {
                  clId    :: !Int,
                  clCount :: !Int,
                  clSum   :: !Vector,
                  clCent  :: !Vector
               }

sqDistance :: Vector -> Vector -> Double
sqDistance (Vector x1 y1) (Vector x2 y2)
  = ((x1-x2)^2) + ((y1-y2)^2)

makeCluster :: Int -> [Vector] -> Cluster
makeCluster clid vecs
  = Cluster { clId = clid,
              clCount = count,
              clSum = vecsum,
              clCent = centre }
  where
   vecsum@(Vector a b)  = foldl' addVector (Vector 0 0) vecs
   centre = Vector (a / fromIntegral count)
                   (b / fromIntegral count)
   count = fromIntegral (length vecs)

-- assign each vector to the nearest cluster centre
assign :: Int -> [Cluster] -> [Vector] -> Array Int [Vector]
assign nclusters clusters points =
  accumArray (flip (:)) [] (0, nclusters-1)
     [ (clId (nearest p), p) | p <- points ]
  where
    nearest p = fst $ minimumBy (compare `on` snd)
                        [ (c, sqDistance (clCent c) p)
                        | c <- clusters ]

-- compute clusters from the assignment
makeNewClusters :: Array Int [Vector] -> [Cluster]
makeNewClusters arr =
  filter ((>0) . clCount) $
     [ makeCluster i ps | (i,ps) <- assocs arr ]

step :: Int -> [Cluster] -> [Vector] -> [Cluster]
step nclusters clusters points =
   makeNewClusters (assign nclusters clusters points)
```

Figure 7: Haskell code for K-Means

```
kmeans_seq :: Int -> [Vector] -> [Cluster] -> IO [Cluster]
kmeans_seq nclusters points clusters = do
  let
      loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany = return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        let clusters' = step nclusters clusters points
        if clusters' == clusters
           then return clusters
           else loop (n+1) clusters'
  --
  loop 0 clusters
```

Figure 8: Haskell code for kmeans_seq

chunk in parallel, somehow combining the results. In order to do this, we need a `combine` function, such that

```
points == as ++ bs
  ==>
step n cs points == step n cs as `combine` step n cs bs
```

Fortunately defining `combine` is not difficult. A cluster is a set of points, from which we can compute a centroid. The intermediate values in this calcuation are the sum and the count of the data points. So a combined cluster can be computed from two independent sub-clusters by taking the sum of these two intermediate values, and re-computing the centroid from them. Since addition is associative and commutative, we can compute sub-clusters in any way we wish and then combine them in this way.

Our Haskell code for combining two clusters is as follows:

```
combineClusters c1 c2 =
  Cluster {clId = clId c1,
           clCount = count,
           clSum = vecsum,
           clCent = Vector (a / fromIntegral count)
                           (b / fromIntegral count)}
  where count = clCount c1 + clCount c2
        vecsum@(Vector a b)  = addVector (clSum c1) (clSum c2)
```

In general, however, we will be processing N chunks of the data space independently, each of which returns a set of clusters. So we need to reduce the N sets of sets of clusters to a single set. This is done with another `accumArray`:

```
reduce :: Int -> [[Cluster]] -> [Cluster]
reduce nclusters css =
  concatMap combine $ elems $
```

```
    accumArray (flip (:)) [] (0,nclusters)
       [ (clId c, c) | c <- concat css]
 where
  combine [] = []
  combine (c:cs) = [foldr combineClusters c cs]
```

Now, the parallel K-Means implementation can be expressed as an application of `parList` to invoke `step` on each chunk, followed by a call to `reduce` to combine the results from the chunks:

```
1  kmeans_par :: Int -> Int -> [Vector] -> [Cluster]
2             -> IO [Cluster]
3  kmeans_par chunks nclusters points clusters = do
4    let chunks = split chunks points
5    let
6        loop :: Int -> [Cluster] -> IO [Cluster]
7        loop n clusters | n > tooMany = return clusters
8        loop n clusters = do
9          hPrintf stderr "iteration %d\n" n
10         hPutStr stderr (unlines (map show clusters))
11         let
12              new_clusterss =
13                 map (step nclusters clusters) chunks
14                    `using` parList rdeepseq
15
16             clusters' = reduce nclusters new_clusterss
17
18         if clusters' == clusters
19            then return clusters
20            else loop (n+1) clusters'
21   --
22   loop 0 clusters
```

the only difference from the sequential implementation is at lines 11–14, where we map `step` over the chunks applying the `parList` strategy, and then call `reduce`.

Note that there's no reason the number of chunks has to be related to the number of processors; as we saw earlier, it is better to produce plenty of sparks and let the runtime schedule them automatically, since this should enable the program to scale over a wide range of processors.

Figure 9 shows the speedups obtained by this implementation for a randomly-generated data set consisting of 4 clusters with a total of approximately 170000 points in 2-D space. The data was generated using the Haskell `normaldistribution` package in order to generate realistically clustered points[11]. For this benchmark we used 1000 for the `chunk` parameter to `kmeans_par`.

[11]The program used to generate the data is provided as `kmeans/GenSamples.hs` in the sample code distribution, and the sample data we used for this benchmark is provided in the files `kmeans/points.bin` and `kmeans/clusters` (the `GenSamples` program will overwrite these files, so be careful if you run it!)

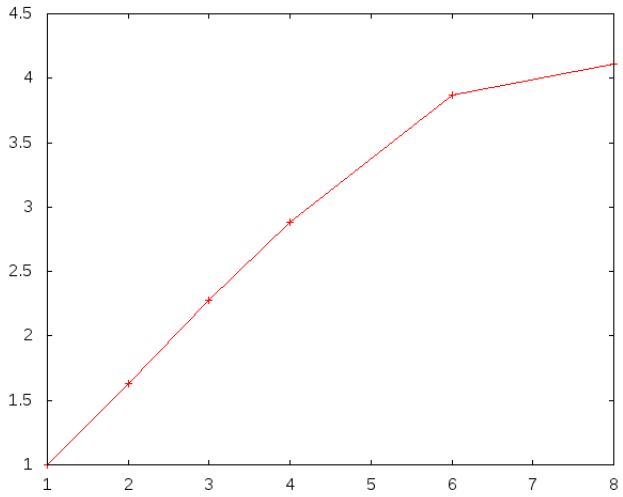


Figure 9: Scaling of parallel K-Means

The results show the algorithm scaling reasonably well up to 6 cores, with a drop in performance at 8 cores. We leave it as an exercise for the reader to analyse the performance and improve it further!

5.3 Further Reading

We have barely scratched the surface of the possibilities with the `Eval` monad and Strategies here. Topics that we have not covered include:

- Sequential strategies, which allow greater control over the specification of *evaluation degree* than is provided by `rseq` and `rdeepseq`. See the documentation for the `Control.Seq` module[12].
- Clustering, which allows greater control over granularity.
- `parBuffer`: a combinator for parallelising lazy streams.

To learn more, we recommend the following resources:

- The documentation for the `Control.Parallel.Strategies` module[13].
- Marlow et al. [8], which explains the motivation behind the design and implementation of `Eval` and Strategies.

[12] `http://hackage.haskell.org/packages/archive/parallel/3.1.0.1/doc/html/Control-Seq.html`

[13] `http://hackage.haskell.org/packages/archive/parallel/3.1.0.1/doc/html/Control-Parallel-Strategies.html`

- Peyton Jones and Singh [13], an earlier tutorial covering basic parallelism in Haskell (beware: this dates from before the introduction of the `Eval` monad).
- Trinder et al. [14], which has a wide range of examples. However beware: this paper is based on the earlier version of Strategies, and some of the examples may no longer work due to the new GC behaviour on sparks; also some of the names of functions and types in the library have since changed.

6 Dataflow parallelism: the `Par` monad

Sometimes there is a need to be *more explicit* about dependencies and task boundaries than it is possible to be with `Eval` and Strategies. In these cases the usual recourse is to Concurrent Haskell, where we can fork threads and be explicit about which thread does the work. However, that approach throws out the baby with the bathwater: determinism is lost. The programming model we introduce in this section fills the gap between Strategies and Concurrent Haskell: it is explicit about dependencies and task boundaries, but without sacrificing determinism. Furthermore the programming model has some other interesting benefits: for example, it is implemented entirely as a Haskell library and the implementation is readily modified to accommodate alternative scheduling strategies.

As usual, the interface is based around a monad, this time called `Par`:

```
newtype Par a
instance Functor Par
instance Applicative Par
instance Monad Par

runPar :: Par a -> a
```

As with the `Eval` monad, the `Par` monad returns a pure result. However, use `runPar` with care: internally it is much more expensive than `runEval`, because (at least in the current implementation) it will fire up a new scheduler instance consisting of one worker thread per processor. Generally speaking the program should be using `runPar` to schedule large-sale parallel tasks.

The purpose of `Par` is to introduce parallelism, so we need a way to create parallel tasks:

```
fork :: Par () -> Par ()
```

`fork` does exactly what you would expect: the computation passed as the argument to `fork` (the "child") is executed concurrently with the current computation (the "parent").

Of course, `fork` on its own isn't very useful; we need a way to communicate results from the child of `fork` to the parent, or in general between two

parallel `Par` computations. Communication is provided by the `IVar` type[14] and its operations:

```
data IVar a  -- instance Eq

new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

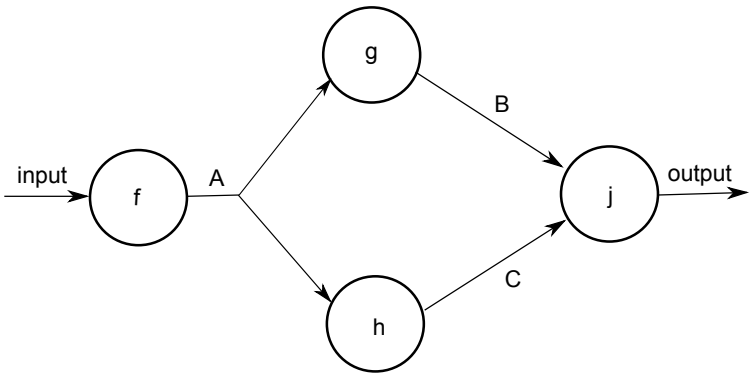
`new` creates a new `IVar`, which is initially empty; `put` fills an `IVar` with a value, and `get` retrieves the value of an `IVar` (waiting until a value has been `put` if necessary). Multiple `put`s to the same `IVar` result in an error.

The `IVar` type is a relative of the `MVar` type that we shall see later in the context of Concurrent Haskell (Section 9), the main difference being that an `IVar` can only be written once. An `IVar` is also like a *future* or *promise*, concepts that may be familiar from other parallel or concurrent languages.

Together, `fork` and `IVar`s allow the construction of *dataflow* networks. The nodes of the network are created by `fork`, and edges connect a `put` with each `get` on that `IVar`. For example, suppose we have the following four functions:

```
f :: In -> A
g :: A -> B
h :: A -> C
j :: (B,C) -> Out
```

Composing these functions forms the following dataflow graph:



There are no sequential dependencies between `g` and `h`, so they could run in parallel. In order to take advantage of the parallelism here, all we need to do is express the graph in the `Par` monad:

```
do
   [ia,ib,ic] <- replicateM 4 new

   fork $ do x <- get input
```

[14] `IVar` is so-called because it is an implementation of I-Structures, a concept from the Parallel Haskell variant pH

```
            put ia (f x)

  fork $ do a <- get ia
            put ib (g a)

  fork $ do a <- get ia
            put ic (h a)

  fork $ do b <- get ib
            c <- get ic
            put output (j b c)
```

For each edge in the graph we make an `IVar` (here `ia`, `ib` and so on). For each node in the graph we call `fork`, and the code for each node calls `get` on each input, and `put` on each output of the node. The order of the `fork` calls is irrelevant — the `Par` monad will execute the graph, resolving the dependencies at runtime.

While the `Par` monad is particularly suited to expressing dataflow networks, it can also express other common patterns too. For example, we can build an equivalent of the `parMap` combinator that we saw earlier in Section 4. First, we build a simple abstraction for a parallel computation that returns a result:

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

The `spawn` function forks a computation in parallel, and returns an `IVar` that can be used to wait for the result.

Now, parallel map consists of calling `spawn` to apply the function to each element of the list, and then waiting for all the results:

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

Note that there are a couple of differences between this and the `Eval` monad `parMap`. First, the function argument returns its result in the `Par` monad; of course it is easy to lift an arbitrary pure function to this type, but the monadic version allows the computation on each element to produce more parallel tasks, or augment the dataflow graph in other ways. Second, `parMapM` waits for all the results. Depending on the context, this may or may not be the most useful behaviour, but of course it is easy to define the other version if necessary.

6.1 A parallel type inferencer

In this section we will parallelise a type inference engine using the `Par` monad. Type inference is a natural fit for the dataflow model, because we can consider each binding to be a node in the graph, and the edges of the graph carry inferred types from bindings to usage sites in the program.

For example, consider the following set of bindings that we want to infer types for:

```
f = ...
g = ... f ...
h = ... f ...
j = ... g ... h ...
```

This pattern gives rise to a dataflow graph with exactly the shape of the example 4-node graph in the previous section: after we have inferred a type for `f`, we can use that type to infer types for `g` and `h` (in parallel), and once we have the types for `g` and `h` we can infer a type for `j`.

Building a dataflow graph for the type inference problem allows the maximum amount of parallelism to be extracted from the type inference process. The actual amount of parallelism present depends on the structure of the input program, however.

The parallel type inferencer can be found in the directory `parinfer` of the code samples, and is derived from a (rather ancient) type inference engine written by Phil Wadler. The types from the inference engine that we will need to work with are as follows:

```
type VarId = String -- variables

data Env -- environment for the type inferencer

-- build environments
makeEnv :: [(VarId,Type)] -> Env

data MonoType -- monomorphic types
data PolyType -- polymorphic types

-- Terms in the input program
data Term = Let VarId Term Term | ...
```

The input to this type inferencer is a single `Term` which may contain `let` bindings, and so to parallelise it we will strip off the outer `let` bindings and typecheck them in parallel. The inner term will be typechecked using the ordinary sequential inference engine. We could have a more general parallel type inference algorithm by always typechecking a `let` binding in parallel with the body, rather than just for the outer `let`s, but that would require threading the `Par` monad through the type inference engine, so for this simple example we are only parallelising inference for the outer bindings.

We need two functions from the inference engine. First, a way to infer a polymorphic type for the right-hand side of a binding:

```
inferTopRhs :: Env -> Term -> PolyType
```

and secondly, a way to run the inference engine on an arbitrary term:

```
inferTopTerm :: Env -> Term -> MonoType
```

The basic idea is that while the sequential inference engine uses an `Env` that maps `VarId`s to `PolyType`s, the parallel part of the inference engine will use an environment that maps `VarId`s to `IVar PolyType`, so that we can `fork` the inference engine for a given binding, and then wait for its result later[15]. The environment for the parallel type inferencer is called `TopEnv`:

```
type TopEnv = Map VarId (IVar PolyType)
```

All that remains is to write the top-level loop. We will write a function `inferTop` with the following type:

```
inferTop :: TopEnv -> Term -> Par MonoType
```

There are two cases to consider. First, when we are looking at a `let` binding:

```
1  inferTop topenv (Let x u v) = do
2      vu <- new
3
4      fork $ do
5        let fu = Set.toList (freeVars u)
6        tfu <- mapM (get . fromJust . flip Map.lookup topenv) fu
7        let aa = makeEnv (zip fu tfu)
8        put vu (inferTopRhs aa u)
9
10     inferTop (Map.insert x vu topenv) v
```

On line 2 we create a new `IVar` `vu` to hold the type of `x`. Lines 4–8 implement the typechecking for the binding:

4 We `fork` here, so that the binding is typechecked in parallel,

5 Find the `IVar`s corresponding to the free variables of the right-hand side

6 Call `get` for each of these, thus waiting for the typechecking of the binding corresponding to each free variable

7 Make a new `Env` with the types we obtained on line 6

8 Call the type inferencer for the right-hand side, and put the result in the `IVar` `vu`.

[15] We are ignoring the possibility of type errors here; in a real implementation the `IVar` would probably contain an `Either` type representing either the inferred type or an error.

The main computation continues (line 10) by typechecking the body of the `let` in an environment in which the bound variable `x` is mapped to the `IVar vu`.

The other case of `inferTop` handles all other expression constructs:

```
1 inferTop topenv t = do
2     let (vs,ivs) = unzip (Map.toList topenv)
3     tvs <- mapM get ivs
4     let aa = makeEnv (zip vs tvs)
5     return (inferTopTerm aa t)
```

This case is straightforward: just call `get` to obtain the inferred type for each binding in the `TopEnv`, construct an `Env`, and call the sequential inferencer on the term `t`.

This parallel implementation works quite nicely. For example, we have constructed a synthetic input for the type checker, a fragment of which is given below (the full version is in the file `code/parinfer/example.in`). The expression defines two sequences of bindings which can be inferred in parallel. The first sequence is the set of bindings for `x` (each successive binding for `x` shadows the previous), and the second sequence is the set of bindings for `y`. Each binding for `x` depends on the previous one, and similarly for the `y` bindings, but the `x` bindings are completely independent of the `y` bindings. This means that our parallel typechecking algorithm should automatically infer types for the `x` bindings in parallel with the inference of the `y` bindings, giving a maximum speedup of 2.

```
let id = \x.x in
    let x = \f.f id id in
    let x = \f . f x x in
    let x = \f . f x x in
    let x = \f . f x x in
    ...
    let x = let f = \g . g x in \x . x in
    let y = \f.f id id in
    let y = \f . f y y in
    let y = \f . f y y in
    let y = \f . f y y in
    ...
    let y = let f = \g . g y in \x . x in
    \f. let g = \a. a x y in f
```

When we type check this expression with one processor, we obtain the following result:

```
$ ./infer <./example.in +RTS -s
  ...
  Total time    1.13s  (  1.12s elapsed)
```

and with two processors:

```
$ ./infer <./example.in +RTS -s -N2
  ,..
  Total time    1.19s  (  0.60s elapsed)
```

representing a speedup of 1.87.

6.2 The `Par` monad compared to Strategies

We have presented two different parallel programming models, each with advantages and disadvantages. Below we summarise the trade-offs so that you can make an informed decision for a given task as to which is likely to be the best choice:

- Using Strategies and the `Eval` monad requires some understanding of the workings of lazy evaluation. Newcomers often find this hard, and diagnosing problems can be difficult. This is part of the motivation for the `Par` monad: it makes all dependencies explicit, effectively replacing lazy evaluation with explicit `put`/`get` on `IVar`s. While this is certainly more verbose, it is less fragile and easier to work with.

  Programming with `rpar` requires being careful about retaining references to sparks to avoid them being garbage collected; this can be subtle and hard to get right in some cases. The `Par` monad has no such requirements, although it does not support speculative parallelism in the sense that `rpar` does: speculative paralelism in the `Par` monad is always executed.

- Strategies allow a separation between algorithm and parallelism, which allows more reuse in some cases.

- The `Par` monad requires threading the monad throughout a computation which is to be parallelised. For example, to parallelise the type inference of all `let` bindings in the example above would have required threading the `Par` monad through the inference engine (or adding `Par` to the existing monad stack), which might be impractical. `Par` is good for localised parallelism, whereas Strategies can be more easily used in cases that require parallelism in multiple parts of the program.

- The `Par` monad has more overhead than the `Eval` monad, although there is no requirement to rebuild data structures as in `Eval`. At the present time, `Eval` tends to perform better at finer granularities, due to the direct runtime system support for sparks. At larger granularities, `Par` and `Eval` perform approximately the same.

- The `Par` monad is implemented entirely in a Haskell library (the `monad-par` package), and is thus readily modified should you need to.

7 GPU programming with *Accelerate*

This section will be a brief introduction to using the *Accelerate* framework for GPU programming.

Over the next few sections we will be introducing the various concepts of *Accelerate*, illustrated by examples that you can type directly into GHCi. These small examples will not be running on the GPU, instead they will be using *Accelerate*'s interpreter. To play with examples yourself, first make sure the `accelerate` package is installed:

```
$ cabal install accelerate
```

The `accelerate` package provides the basic infrastructure: the module `Data.Array.Accelerate` for constructing array computations, and `Data.Array.Accelerate.Interp` for interpreting them. To actually run an *Accelerate* computation on a GPU you will also need the `accelerate-cuda` package, but we'll cover that later (Section 7.10).

Next, start up GHCi and import the necessary modules:

```
$ ghci
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I
Prelude A I>
```

7.1 Overview

Accelerate is a *domain-specific language* for programming GPUs. Programs are written in Haskell syntax using operations of the framework, but the method by which the program runs is very different from a conventional Haskell program. Broadly speaking, a program fragment that uses *Accelerate* works like this:

- The Haskell code generates a data structure in an internal representation that the programmer doesn't get to see,
- This data structure is then either *compiled* into GPU code and run directly on the GPU, or it can be *interpreted* using *Accelerate*'s built-in interpreter. Both methods give the same results, but of course running on the GPU should be far faster.

By the magic of Haskell's overloading and abstraction facilities, the Haskell code that you write using *Accelerate* usually looks very much like ordinary Haskell code, even though it is generating a data structure rather than actually evaluating the result directly.

While reading this tutorial you probably want to have a copy of the *Accelerate* API documentation to hand: `http://hackage.haskell.org/`

packages/archive/accelerate/0.12.0.0/doc/html/Data-Array-Accelerate.html.

7.2 Arrays and indices

Everything in *Accelerate* revolves around arrays. Arrays are the fundamental datatype that we operate on: an *Accelerate* computation takes arrays as inputs and delivers one or more arrays as output.

The type of arrays has two parameters:

```
data Array sh e
```

where `e` is the element type. The `sh` parameter describes the *shape* of the array, that is, the number of dimensions. Unlike the `Ix` class that standard Haskell arrays are paramterised over, the shape parameter is rather more flexible, as we shall see.

Shapes are built out of two type constructors, `Z` and `:.`

```
data Z = Z
data tail :. head = tail :. head
```

For example, `Z` is the shape of an array with no dimensions, i.e. a scalar, which has a single element. If we add a dimension, `Z :. Int`, this is the shape of an array with a single dimension indexed by `Int`, otherwise known as a vector. Adding another dimension gives `Z :. Int :. Int`, the shape of two-dimensional arrays. Note that new dimensions are added on the right, and the rightmost index is the one that "varies the fastest".

Remember that `Z` and `:.` are both type constructors and value constructors; this can get a bit confusing at times. For example, `Z :. 3` has type `Z :. Int`. The value form is used in *Accelerate* to mean either "sizes" or "indices"; for example, `Z :. 3` can be either the shape of 3-element vectors, or the index of the third element of a vector.

A few handy type synonyms are provided for the common shape types:

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
```

In fact, *Accelerate* only supports `Int`-typed indices, and indices always begin at zero. Since dimensionalities of zero and one are common, the library provides type synonyms for those:

```
type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
```

The library allows us to build arrays and experiment with them in ordinary Haskell code, so let's try a few examples. A simple way to build an array is using `fromList`:

```
fromList :: (Shape sh, Elt e) => sh -> [e] -> Array sh e
```

Don't worry about the `Shape` and `Elt` type classes, they are just there to ensure that we only use the approved shape constructors (`Z` and `:.`) and approved element types respectively.

Let's build a 10-element vector of `Int`, and fill it with the numbers $1 \ldots 10$. We need to pass a shape argument, which will be `Z:.10` for a 10-element vector:

```
> fromList (Z:.10) [1..10]
<interactive>:9:1:
    No instance for (Shape (Z :. head0))
      arising from a use of 'fromList'
    Possible fix: add an instance declaration for (Shape (Z :. head0))
    In the expression: fromList (Z :. 10) [1 .. 10]
    In an equation for 'it': it = fromList (Z :. 10) [1 .. 10]
```

Oops! This illustrates something that you will probably encounter a lot when working with *Accelerate*: a type error caused by insufficient type information. In this case, since integers are overloaded in Haskell, we have to say explicitly that we mean `Int` for the indices of our vector. There are many ways to give GHC the extra bit of information it needs, one way is to add a `Vector Int` type signature to the whole expression:

```
> fromList (Z:.10) [1..10] :: Vector Int
Array (Z :. 10) [1,2,3,4,5,6,7,8,9,10]
```

Similarly, we can make a two-dimensional array, with 3 rows of 5 columns:

```
> fromList (Z:.3:.5) [1..] :: Array DIM2 Int
Array (Z :. 3 :. 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Even though we've created a two-dimensional array, we initialised it from a flat list, and it still looks like a flat list when printed out. In fact, all arrays are represented as flat lists (after all, computer memory is one-dimensional). The shape of the array is only there to tell the library how to interpret the operations on it—if we ask for the index `Z:.2:.1` in an array with shape `Z:.3:.5` we'll get the element at position $2 \times 5 + 1$. We can try it using `indexArray`:

```
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> indexArray arr (Z:.2:.1)
12
```

One thing to remember is that in *Accelerate*, arrays cannot be nested: it is not possible to build an array of arrays. This is because arrays must be able to be mapped directly into flat arrays on the GPU, which has no support for nested arrays.

We can, however, have arrays of tuples. For example:

```
> fromList (Z:.2:.3) (Prelude.zip [1..] [1..]) :: Array DIM2 (Int,Int)
Array (Z :. 2 :. 3) [(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)]
```

Internally, *Accelerate* will translate an array of tuples into a tuple of arrays; this is done entirely automatically and we don't need to worry about it. Arrays of tuples are a very useful structure as we shall see.

7.3 Running a simple *Accelerate* computation

So far we have been experimenting with arrays in the context of ordinary Haskell code, we haven't constructed an actual *Accelerate* computation over arrays yet. An *Accelerate* computation takes the form `run` E, where

```
run :: Arrays a => Acc a -> a
```

and E is an expression of type `Acc a`, which means "an accelerated computation that delivers a value of type `a`". The `Arrays` class allows `a` to be either an array, or a tuple of arrays. A value of type `Acc a` is really a data structure (we'll see in a moment how to build it), and the `run` function evaluates the data structure to produce a result. There are two variants of `run`: one exported by `Data.Array.Accelerate.Interpreter` that we will be using for experimentation and testing, and another exported by `Data.Array.Accelerate.CUDA` (in the `accelerate-cuda` package) that runs the computation on the GPU.

Let's try a very simple example. Starting with the $3x5$ array of `Int` from the previous section, we will add one to every element:

```
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ A.map (+1) (use arr)
Array (Z :. 3 :. 5) [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

Breaking this down, first we call `A.map`, which is the function that maps over arrays (we needed the `A.` prefix to disambiguate with `Prelude.map`; recall that we used `import Data.Array.Accelerate as A` earlier).

```
A.map ::
  (Shape ix, Elt a, Elt b) =>
  (Exp a -> Exp b) -> Acc (Array ix a) -> Acc (Array ix b)
```

As its second argument, `A.map` takes an `Acc (Array ix a)`, whereas we have an `Array DIM2 Int`. In order to get this array into the `Acc` world, we have to apply the function `use`:

```
use :: Arrays arrays => arrays -> Acc arrays
```

`use` is the way to take arrays from Haskell and inject them into an *Accelerate* computation. When we run on the GPU, this might actually involve copying the array from the computer's main memory into the GPU's local memory.

The first argument to `A.map` has type `Exp a -> Exp b`. Here `Exp` is a bit like `Acc`: it represents a computation in the world of *Accelerate*, but whereas `Acc` is a computation on arrays, `Exp` is a computation on single values.

As a rule of thumb, in `Exp a` the `a` is always an instance of the type class `Elt`, which is the class of types allowed to be array elements. `Elt` includes all the usual numeric types, and also indices and tuples. In `Acc a`, the `a` is always an instance of the type class `Arrays`, which as we saw earlier includes only arrays and tuples of arrays.

The reason that *Accelerate* separates `Exp` and `Acc` is to enforce the no-nesting policy: the argument to `A.map` can only be a function on array elements, not arrays.

In the example we passed `(+1)` as the first argument to map. How does this have type `Exp a -> Exp b`? Normally we would expect it to have type `Num a => a -> a`, because `+` is a method in the `Num` class. The `accelerate` package provides an instance for `Num (Exp a)`[16] so numeric expressions built using the usual overloaded numeric operations in Haskell work just fine in the world of `Exp`. Indeed the expression also involved the constant `1`, and since constants in Haskell are overloaded, *Accelerate* provides an instance for `Integral` that lifts the constant into `Exp`.

Here's another example, squaring every element in the array:

```
> run $ A.map (^2) (use arr)
Array (Z :. 3 :. 5) [1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]
```

Remember, we can't write arbitrary operations in `Exp`. There's no way in general to get from an `Exp a` to `a`, and the operations on `Exp` are strictly limited to those provided by the library. This makes sense, since `Exp` is really a data structure that will be compiled to GPU code (which is like C), so `Exp` can only support the operations that are available in the target language.

7.4 Folds over arrays

A common operation on arrays is to *fold* the array. As you might expect, *Accelerate* provides a `fold` operation that works in the natural way:

```
> let arr = fromList (Z:.10) [1..10] :: Vector Int
> run $ fold (+) 0 (use arr)
Array (Z) [55]
```

Here we folded the operation `(+)` over the array, using `0` as the initial element, and the result is the sum of the elements in the array. Note that the result was in the form of a single-element array, i.e. a scalar. This is because Folding in *Accelerate* has an interesting type:

[16] with a couple of extra constraints, which we won't go into here.

```
fold :: (Shape ix, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a -> Acc (Array (ix :. Int) a) -> Acc (Array ix a)
```

`fold` takes an array that has at least one dimension, and returns an array *with one less dimension*. In other words, it is polymorphic in the shape of the input array. On a multi-dimensional array, `fold` folds over the rightmost dimension. For example, in a two-dimensional array, the `fold` will be applied to each row separately:

```
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ fold (+) 0 (use arr)
Array (Z :. 3) [15,40,65]
```

There is one other important aspect of `fold` to bear in mind: it is neither a left nor a right fold, in fact it is an associative fold. This is necessary in order to be able to execute it in parallel on the GPU. This does mean that the operation you pass as the first argument of `fold` must be associative, otherwise the results are unpredictable.[17]

7.5 More operations on arrays

```
(!) :: (Shape ix, Elt e) => Acc (Array ix e) -> Exp ix -> Exp e
```

```
zipWith :: (Shape ix, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array ix a) -> Acc (Array ix b)
        -> Acc (Array ix c)
```

```
reshape :: (Shape ix, Shape ix', Elt e)
        => Exp ix -> Acc (Array ix' e)
        -> Acc (Array ix e)
```

7.6 Creating arrays inside `Acc`

```
fill :: (Shape sh, Elt e)
     => Exp sh -> Exp e
     -> Acc (Array sh e)
```

```
generate :: (Shape ix, Elt a)
         => Exp ix -> (Exp ix -> Exp a)
         -> Acc (Array ix a)
```

7.7 `Acc` and `Exp`, lifting and unlifting

We saw above that simple integer literals and numeric operations are automatically operations in `Exp` by virtue of being overloaded. This isn't always

[17] strictly speaking it should be called `unsafeFold`, since to be used safely requires the programmer to obey a precondition not expressed in the types.

the case—sometimes we need to perform some explicit conversion in order to build operations in `Exp` and `Acc`, or to convert between the two.

What if we already have a value of type `Int` and we want an `Exp Int`? This is what the function `constant` is for:

```
constant :: Elt t => t -> Exp t
```

Note that `constant` only works for things of type `Elt`, which you may recall is the class of types allowed to be array elements, including numeric types, indices and tuples of `Elt`s.

Scalar arrays If we have an `Exp e`, we can turn it into a `Scalar` using `unit`:

```
unit :: Elt e => Exp e -> Acc (Scalar e)
```

and the dual to this is `the`:

```
the :: Elt e => Acc (Scalar e) -> Exp e
```

7.7.1 Lifting and unlifting

7.8 Slicing arrays

ToDo:

7.9 Boolean operations

ToDo:

```
(?) :: Elt t => Exp Bool -> (Exp t, Exp t) -> Exp t
(==*) :: (Elt t, IsScalar t) => Exp t -> Exp t -> Exp Bool
```

7.10 Running on the GPU

```
$ cabal install accelerate-cuda -fdebug
```

7.11 A Mandelbrot Set generator

In this section we put together what we have learned so far and build a mandelbrot set generator that runs on the GPU. The end result will be the picture in Figure 7.11.

The Mandelbrot Set is a mathematical construction over the *complex plane*, that is the two-dimensional plane of complex numbers. A particular point is said to be in the set if when the following equation is repeatedly applied, $|z|$ does not diverge to infinity:

$$z_{n+1} = c + z_n^2$$

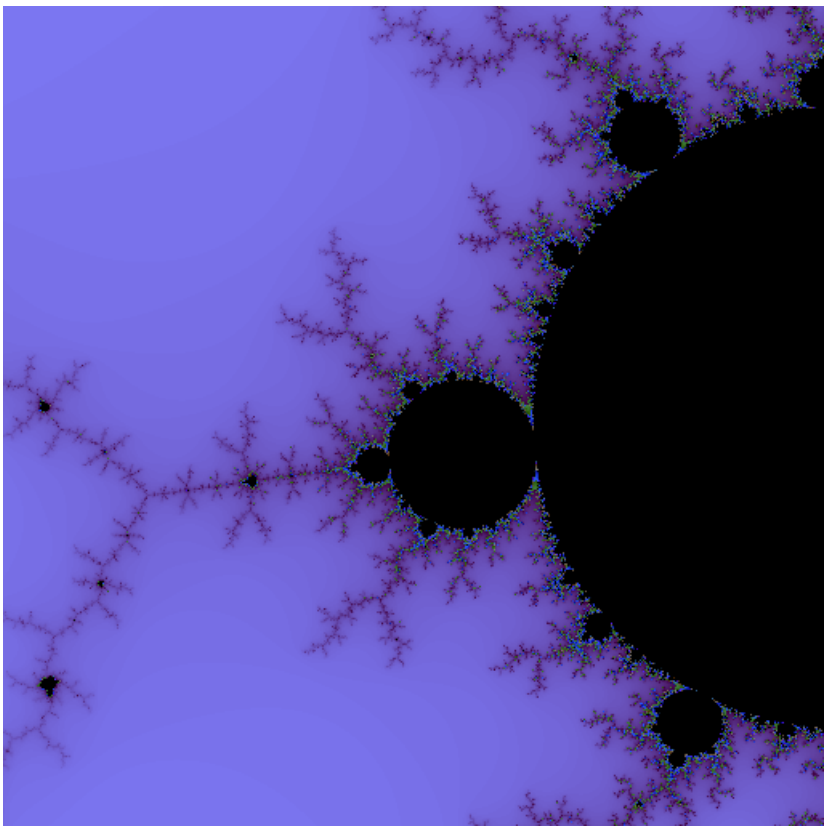


Figure 10: Mandelbot Set picture generated on the GPU

where c is the point on the plane (a complex number), and $z_0 = c$.

In practice we iterate the equation for a fixed number of times, and if it has not diverged at that point, then we declare the point to be in the set. Furthermore, to generate a pretty picture, we remember the iteration at which each point diverged and map the iteration values to some colour palette.

We know that $|z|$ will definitely diverge if it is greater than 2. Now the magnitude of a complex number $x + iy$ is given by $\sqrt{x^2 + y^2}$, so we can simplify the condition by squaring both sides, giving us that divergence happens when $x^2 + y^2 > 4$.

Let's express this using *Accelerate*[18]. First we want a type for complex numbers. *Accelerate* lets us work with tuples, so we can represent complex numbers as pairs of floating-point numbers. As it happens, not all GPUs can work with `Doubles`, so we'll use `Float`:

```
type F       = Float
type Complex = (F,F)
```

We'll be referring to `Float` a lot, so the `F` type synonym helps to keep things readable. Now, to calculate the next value of z at a given point c looks like this:

```
next :: Exp Complex -> Exp Complex -> Exp Complex
next c z = c `plus` (z `times` z)
```

We can't use the normal `+` and `*` operations here, because there is no instance of `Num` for `Exp Complex`—*Accelerate* doesn't know how to add or multiply our complex numbers, so we have to define these operations ourselves. First, `plus`:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = ...
```

To sum two complex numbers we need to just sum the components. But how can we access the components? We cannot pattern match on `Exp Complex`. There are a few different ways to do it, and we'll explore them briefly. *Accelerate* provides operations for selecting the components of pairs in `Exp`, namely:

```
fst :: (Elt a, Elt b) => Exp (a, b) -> Exp a
snd :: (Elt a, Elt b) => Exp (a, b) -> Exp b
```

So we could write plus like this:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = ...
  where
    ax = A.fst a
    ay = A.snd a
    bx = A.fst b
    by = A.snd b
```

[18] the full sample code is in `mandel/mandel.hs`

but how do we construct the result? We want to write something like `(ax+bx, ay+by)`, but this has type `(Exp F, Exp F)`, whereas we want `Exp (F,F)`. Fortunately there is a function that can perform exactly this conversion: `lift` (see Section 7.7.1). So the result is:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = lift (ax+bx, ay+by)
  where
    ax = A.fst a
    ay = A.snd a
    bx = A.fst b
    by = A.snd b
```

In fact we could do a little better, since `A.fst` and `A.snd` are just instances of `unlift`, and we could do them both in one go:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = lift (ax+bx, ay+by)
  where
    (ax, ay) = unlift a
    (bx, by) = unlift b
```

although unfortunately if you try this you will find that there isn't enough type information for GHC, so we have to help it out a bit:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = lift (ax+bx, ay+by)
  where
    (ax, ay) = unlift a :: (Exp F, Exp F)
    (bx, by) = unlift b :: (Exp F, Exp F)
```

We can go a little further, since *Accelerate* provides a useful function that incorporates both the `lift` and `unlift`. For a 2-argument function, the right variant is called `lift2`:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus = lift2 f
  where f :: (Exp F, Exp F) -> (Exp F, Exp F) -> (Exp F, Exp F)
        f (ax,ay) (bx,by) = (ax+bx,ay+by)
```

Unfortunately again we had to add the type signature to get it to typecheck, but it does aid readability. This is perhaps as close to "natural" as we can get for this definition: the necessary lifting and unlifting are confined to just one place.

We also need to define `times`, which follows the same pattern as `plus`, although of course this time we are multiplying the two complex numbers together:

```
times :: Exp Complex -> Exp Complex -> Exp Complex
times = lift2 f
  where f :: (Exp F, Exp F) -> (Exp F, Exp F) -> (Exp F, Exp F)
        f (ax,ay) (bx,by)   =  (ax*bx-ay*by, ax*by+ay*bx)
```

So now we can compute z_{n+1} given z and c. But we need to think about the program as a whole: for each point, we need to iterate this process until

divergence, and then remember the number of iterations at which divergence happened. This creates a small problem: GPUs are designed to do the *same thing* to lots of different data at the same time, whereas we want to do something different depending on whether or not a particular point has diverged or not. So in practice we can't do what we would normally do in a single-threaded language and iterate each point until divergence, instead we have to find a way to apply the same operation to every element of the array, for a fixed number of iterations.

In fact this is a common conundrum faced by the GPU programmer. As a rule of thumb, conditionals are "bad", because they cause *SIMD divergence*. This means that when the GPU hits a conditional instruction, it first runs all the threads that take the true branch, and then runs the threads that take the false branch. Of course if you have nested conditionals, the amount of parallelism rapidly disappears.

We can't avoid *some* kind of conditional in the mandelbrot example, but we can make sure there is only a bounded amount of divergence by having just one conditional per iteration, and a fixed number of iterations. The trick we use is to keep a pair `(z,i)` for every array element, where `i` is the iteration at which that point diverged. At each iteration:

- Compute `z' = next c z`,
- If it is greater than 4, then the result is `(z,i)`,
- otherwise the result is `(z',i+1)`

Accelerate doesn't allow nested tuples, so we have to flatten the pair `z` in `(z,i)`, to get `(x,y,i)`. The operation we will apply to each element therefore has this type:

```
iter :: Exp Complex -> Exp (F,F,Int) -> Exp (F,F,Int)
iter c z = ...
```

The first thing to do is `unlift z`, so that we can access the components of the triple, and then compute `z'` by calling `next`:

```
  let
     (x,y,i) = unlift z :: (Exp F, Exp F, Exp Int)
     z' = next c (lift (x,y))
  in
```

We had to use `lift` to get `(x,y)` back into the right form for passing to `next`. Now we have `z'` we can do the conditional test:

```
  (dot z' >* 4.0) ?
     ( lift (x,y,i)
     , lift (A.fst z', A.snd z', i+1)
     )
```

where `dot` computes $x^2 + y^2$; it follows the same pattern as `plus` and `times` so we've omitted it here. We are using the infix `?` operator for the conditional (Section 7.9). In the true case, we just return the original `(x,y,i)` suitably lifted, whereas in the false case we return the new `z'` and `i+1`.

The algorithm will need two arrays: one array of `c` values which will be constant throughout the computation, and a second array of `(z,i)` values which will be recomputed by each iteration. Our arrays are 2-dimensional arrays indexed by pixel coordinates, since the aim is to generate a picture from the iteration values at each pixel.

The initial complex plane of `c` values is generated by a function `genPlane`:

```
genPlane :: F -> F    -- X bounds of the view
         -> F -> F    -- Y bounds of the view
         -> Int       -- X resolution in pixels
         -> Int       -- Y resolution in pixels
         -> Acc ComplexPlane
```

its definition is rather long so we omit it here, but essentially it is a call to `generate` (Section 7.6).

From the initial complex plane we can generate the initial array of `(z,i)` values, which is done by initialising each `z` to the corresponding `c` value, and `i` to zero. In the code this can be found in the `mkinit` function.

Now we can put the pieces together and write the code for the complete algorithm:

```
mandelbrot :: F -> F -> F -> F -- plane coordinates
           -> Int -> Int       -- view resolution
           -> Int              -- maximum iterations
           -> Acc (Array DIM2 (F,F,Int))

mandelbrot x y x' y' screenX screenY depth
  = iterate go (mkinit cs) !! depth
  where
    cs  = genPlane x y x' y' screenX screenY

    go :: Acc (Array DIM2 (F,F,Int))
       -> Acc (Array DIM2 (F,F,Int))
    go = A.zipWith iter cs
```

`cs` is our static complex plane generated by `genPlane`. The function `go` performs one iteration, producing a new array of `(z,i)`, and it is expressed by zipping `iter` over both `cs` and the current array of `(z,i)`. To perform all the iterations, we simply call the ordinary list function `iterate`:

```
iterate :: (a -> a) -> a -> [a]
```

and take the element at position `depth`, which corresponds to the `go` function having been applied `depth` times.

Part II
Concurrent Haskell

Concurrent Haskell [11] is an extension to Haskell 2010 [9] adding support for explicitly threaded concurrent programming. The basic interface remains largely unchanged in its current implementation, although a number of embellishments have since been added, which we will cover in later sections:

- Asynchronous exceptions [3] were added as a means for asynchronous cancellation of threads,
- Software Transactional Memory was added [2], allowing safe composition of concurrent abstractions, and making it possible to safely build larger concurrent systems.
- The behaviour of Concurrent Haskell in the presence of calls to and from foreign languages was specified [6]

8 Forking Threads

The basic requirement of concurrency is to be able to fork a new thread of control. In Concurrent Haskell this is achieved with the `forkIO` operation:

```
forkIO :: IO () -> IO ThreadId
```

`forkIO` takes a computation of type `IO ()` as its argument; that is, a computation in the `IO` monad that eventually delivers a value of type `()`. The computation passed to `forkIO` is executed in a new *thread* that runs concurrently with the other threads in the system. If the thread has effects, those effects will be interleaved in an indeterminate fashion with the effects from other threads.

To illustrate the interleaving of effects, let's try a simple example in which two threads are created, once which continually prints the letter `A` and the other printing `B`[19]:

```
1 import Control.Concurrent
2 import Control.Monad
3 import System.IO

5 main = do
6   hSetBuffering stdout NoBuffering
7   forkIO (forever (putChar 'A'))
8   forkIO (forever (putChar 'B'))
9   threadDelay (10^6)
```

[19] this is sample `fork.hs`

Line 6 puts the output `Handle` into non-buffered mode, so that we can see the interleaving more clearly. Lines 7 and 8 create the two threads, and line 9 tells the main thread to wait for one second (`10^6` microseconds) and then exit.

When run, this program produces output something like this:

```
AAAAAAAAABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABAB
```

Note that the interleaving is non-deterministic: sometimes we get strings of a single letter, but often the output switches regularly between the two threads. Why does it switch so regularly, and why does each thread only get a chance to output a single letter before switching? The threads in this example are contending for a single resource: the `stdout` Handle, so scheduling is affected by how contention for this resource is handled. In the case of GHC a Handle is protected by a lock implemented as an `MVar` (described in the next section). We shall see shortly how the implementation of `MVar`s causes the `ABABABA` behaviour.

We emphasised earlier that concurrency is a program structuring technique, or an abstraction. Abstractions are practical when they are efficient, and this is where GHC's implementation of threads comes into its own. Threads are extremely lightweight in GHC: a thread typically costs less than a hundred bytes plus the space for its stack, so the runtime can support literally millions of them, limited only by the available memory. Unlike OS threads, the memory used by Haskell threads is movable, so the garbage collector can pack threads together tightly in memory and eliminate fragmentation. Threads can also expand and shrink on demand, according to the stack demands of the program. When using multiple processors, the GHC runtime system automatically migrates threads between cores in order to balance the load.

User-space threading is not unique to Haskell, indeed many other languages, including early Java implementations, have had support for user-space threads (sometimes called "green threads"). It is often thought that user-space threading hinders interoperability with foreign code and libraries that are using OS threads, and this is one reason that OS threads tend to be preferred. However, with some careful design it is possible to overcome these difficulties too, as we shall see in Section 15.

9 Communication: MVars

The lowest-level communication abstraction in Concurrent Haskell is the `MVar`, whose interface is given below:

```
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

An `MVar` can be thought of as a box that is either empty or full. The `newEmptyMVar` operation creates a new empty box, and `newMVar` creates a new full box containing the value passed as its argument. The `putMVar` operation puts a value into the box, but blocks (waits) if the box is already full. Symmetrically, the `takeMVar` operation removes the value from a full box but blocks if the box is empty.

`MVar`s generalise several simple concurrency abstractions:

- `MVar ()` is a *lock*; `takeMVar` acquires the lock and `putMVar` releases it again.[20] An `MVar` used in this way can protect shared mutable state or critical sections.
- An `MVar` is a one-place channel, which can be used for asynchronous communication between two threads. In Section 9.1 we show how to build unbounded buffered channels from `MVar`s.
- An `MVar` is a useful container for shared mutable state. For example, a common design pattern in Concurrent Haskell when several threads need read and write access to some state, is to represent the state value as an ordinary immutable Haskell data structure stored in an `MVar`. Modifying the state consists of taking the current value with `takeMVar` (which implicitly acquires a lock), and then placing a new value back in the `MVar` with `putMVar` (which implicitly releases the lock again).

We can also use `MVar`s to do some simple asynchronous I/O. Suppose we want to download some web pages concurrently and wait for them all to download before continuing. We are given the following function to download a web page:

```
getURL :: String -> IO ByteString
```

Let's use this to download two URLs concurrently:

```
1  do
2    m1 <- newEmptyMVar
3    m2 <- newEmptyMVar
4
5    forkIO $ do
6      r <- getURL "http://www.wikipedia.org/wiki/Shovel"
7      putMVar m1 r
```

[20] It works perfectly well the other way around too, just be sure to be consistent about the policy.

```
9     forkIO $ do
10      r <- getURL "http://www.wikipedia.org/wiki/Spade"
11      putMVar m2 r
12
13    r1 <- takeMVar m1
14    r2 <- takeMVar m2
15    return (r1,r2)
```

Lines 2–3 create two new empty `MVar`s to hold the results. Lines 5–7 fork a new thread to download the first URL; when the download is complete the result is placed in the `MVar` `m1`, and lines 9–11 do the same for the second URL, placing the result in `m2`. In the main thread, line 13 waits for the result from `m1`, and line 14 waits for the result from `m2` (we could do these in either order), and finally both results are returned.

This code is rather verbose. We could shorten it by using various existing higher-order combinators from the Haskell library, but a better approach would be to extract the common pattern as a new abstraction: we want a way to perform an action *asynchronously*, and later wait for its result. So let's define an interface that does that, using `forkIO` and `MVar`s:

```
1  newtype Async a = Async (MVar a)
2
3  async :: IO a -> IO (Async a)
4  async io = do
5    m <- newEmptyMVar
6    forkIO $ do r <- io; putMVar m r
7    return (Async m)
8
9  wait :: Async a -> IO a
10 wait (Async m) = readMVar m
```

Line 1 defines a datatype `Async` that represents an asynchronous action that has been started. Its implementation is just an `MVar` that will contain the result; creating a new type here might seem like overkill, but later on we will extend the `Async` type to support more operations, such as cancellation.

The `wait` operation uses `readMVar`, defined thus[21]:

```
readMVar :: MVar a -> IO a
readMVar m = do
  a <- takeMVar m
  putMVar m a
  return a
```

that is, it puts back the value into the `MVar` after reading it, the point being that we might want to call `wait` multiple times, or from different threads.

Now, we can use the `Async` interface to clean up our web-page downloading example:

```
1  do
```

[21] `readMVar` is a standard operation provided by the `Control.Concurrent` module

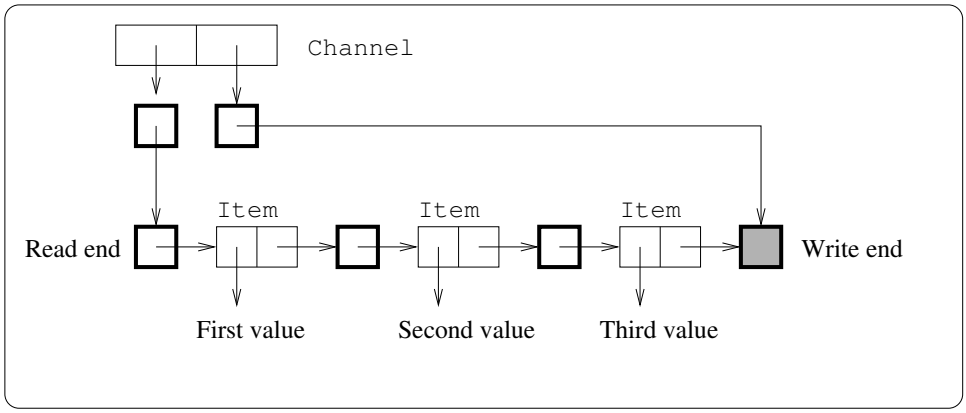


Figure 11: Structure of the buffered channel implementation

```
    a1 <- async $ getURL "http://www.wikipedia.org/wiki/Shovel"
    a2 <- async $ getURL "http://www.wikipedia.org/wiki/Spade"
    r1 <- wait a1
    r2 <- wait a2
    return (r1,r2)
```

Much nicer! To demonstrate this working, we can make a small wrapper that downloads a URL and reports how much data was downloaded and how long it took[22]:

```
sites = ["http://www.google.com",
         "http://www.bing.com",
         ... ]

main = mapM (async.http) sites >>= mapM wait
 where
   http url = do
     (page, time) <- timeit $ getURL url
     printf "downloaded: %s (%d bytes, %.2fs)\n"
        url (B.length page) time
```

which results in something like this:

```
downloaded: http://www.google.com (14524 bytes, 0.17s)
downloaded: http://www.bing.com (24740 bytes, 0.18s)
downloaded: http://www.wikipedia.com/wiki/Spade (62586 bytes, 0.60s)
downloaded: http://www.wikipedia.com/wiki/Shovel (68897 bytes, 0.60s)
downloaded: http://www.yahoo.com (153065 bytes, 1.11s)
```

9.1 Channels

One of the strengths of `MVars` is that they are a useful building block out of which larger abstractions can be constructed. Here we will use `MVars`

[22] the full code can be found in the sample `geturls.hs`

to construct a unbounded buffered channel, supporting the following basic interface:

```
data Chan a

newChan   :: IO (Chan a)
readChan  :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
```

This channel implementation first appeared in Peyton Jones et al. [11] (although the names were slightly different), and is available in the Haskell module `Control.Concurrent.Chan`. The structure of the implementation is represented diagrammatically in Figure 9, where each bold box represents an `MVar` and the lighter boxes are ordinary Haskell data structures. The current contents of the channel are represented as a `Stream`, defined like this:

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
```

The end of the stream is represented by an empty `MVar`, which we call the "hole", because it will be filled in when a new element is added. The channel itself is a pair of `MVar`s, one pointing to the first element of the `Stream` (the read position), and the other pointing to the empty `MVar` at the end (the write position):

```
data Chan a
 = Chan (MVar (Stream a))
        (MVar (Stream a))
```

To construct a new channel we must first create an empty `Stream`, which is just a single empty `MVar`, and then the `Chan` constructor with `MVar`s for the read and write ends, both pointing to the empty `Stream`:

```
newChan :: IO (Chan a)
newChan = do
   hole  <- newEmptyMVar
   readVar  <- newMVar hole
   writeVar <- newMVar hole
   return (Chan readVar writeVar)
```

To add a new element to the channel we must make an `Item` with a new hole, fill in the current hole to point to the new item, and adjust the write-end of the `Chan` to point to the new hole:

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar writeVar
  putMVar writeVar new_hole
  putMVar old_hole (Item val new_hole)
```

To remove a value from the channel, we must follow the read end of the `Chan` to the first `MVar` of the stream, take that `MVar` to get the `Item`, adjust

the read end to point to the next `MVar` in the stream, and finally return the value stored in the `Item`:

```
1 readChan :: Chan a -> IO a
2 readChan (Chan readVar _) = do
3   stream <- takeMVar readVar
4   Item val new <- takeMVar stream
5   putMVar readVar new
6   return val
```

Consider what happens if the channel is empty. The first `takeMVar` (line 3) will succeed, but the second `takeMVar` (line 4) will find an empty hole, and so will block. When another thread calls `writeChan`, it will fill the hole, allowing the first thread to complete its `takeMVar`, update the read end (line 5) and finally return.

If multiple threads concurrently call `readChan`, the first one will successfully call `takeMVar` on the read end, but the subsequent threads will all block at this point until the first thread completes the operation and updates the read end. If multiple threads call `writeChan`, a similar thing happens: the write end of the `Chan` is the synchronisation point, only allowing one thread at a time to add an item to the channel. However, the read and write ends being separate `MVar`s allows concurrent `readChan` and `writeChan` operations to proceed without interference.

This implementation allows a nice generalisation to *multicast* channels without changing the underlying structure. The idea is to add one more operation:

```
dupChan :: Chan a -> IO (Chan a)
```

which creates a duplicate `Chan` with the following semantics:

- The new `Chan` begins empty,
- Subsequent writes to either `Chan` are read from both; that is, reading an item from one `Chan` does not remove it from the other.

The implementation is straightforward:

```
dupChan :: Chan a -> IO (Chan a)
dupChan (Chan _ writeVar) = do
   hole       <- takeMVar writeVar
   putMVar writeVar hole
   newReadVar <- newMVar hole
   return (Chan newReadVar writeVar)
```

Both channels share a single write-end, but they have independent read-ends. The read end of the new channel is initialised to point to the hole at the end of the current contents.

Sadly, this implementation of `dupChan` does not work! Can you see the problem? The definition of `dupChan` itself is not at fault, but combined with

the definition of `readChan` given earlier it does not implement the required semantics. The problem is that `readChan` does not replace the contents of a hole after having read it, so if `readChan` is called to read values from both the channel returned by `dupChan` and the original channel, the second call will block. The fix is to change a `takeMVar` to `readMVar` in the implementation of `readChan`:

```
1 readChan :: Chan a -> IO a
2 readChan (Chan readVar _) = do
3   stream <- takeMVar readVar
4   Item val new <- readMVar stream -- modified
5   putMVar readVar new
6   return val
```

Line 4 returns the `Item` back to the `Stream`, where it can be read by any duplicate channels created by `dupChan`.

Before we leave the topic of channels, consider one more extension to the interface that was described as an "easy extension" and left as an exercise by Peyton Jones et al. [11]:

```
unGetChan :: Chan a -> a -> IO ()
```

the operation `unGetChan` pushes a value back on the read end of the channel. Leaving aside for a moment the fact that the interface does not allow the atomic combination of `readChan` and `unGetChan` (which would appear to be an important use case), let us consider how to implement `unGetChan`. The straightforward implementation is as follows:

```
1 unGetChan :: Chan a -> a -> IO ()
2 unGetChan (Chan readVar _) val = do
3    new_read_end <- newEmptyMVar
4    read_end <- takeMVar readVar
5    putMVar new_read_end (Item val read_end)
6    putMVar readVar new_read_end
```

we create a new hole to place at the front of the `Stream` (line 3), take the current read end (line 4) giving us the current front of the stream, place a new `Item` in the new hole (line 5), and finally replace the read end with a pointer to our new item.

Simple testing will confirm that the implementation works. However, consider what happens when the channel is empty, there is already a blocked `readChan`, and another thread calls `unGetChan`. The desired semantics is that `unGetChan` succeeds, and `readChan` should return with the new element. What actually happens in this case is deadlock: the thread blocked in `readChan` will be holding the read-end `MVar`, and so `unGetChan` will also block (line 4) trying to take the read end. As far as we know, there is no implementation of `unGetChan` that has the desired semantics.

The lesson here is that programming larger structures with `MVar` can be much trickier than it appears. As we shall see shortly, life gets even more

difficult when we consider exceptions. Fortunately there is a solution, that we will describe in Section 11.

Despite the difficulties with scaling `MVar`s up to larger abstractions, `MVar`s do have some nice properties, as we shall see in the next section.

9.2 Fairness

Fairness is a well-studied and highly technical subject, which we do not attempt to review here. Nevertheless, we wish to highlight one particularly important guarantee provided by `MVar`s with respect to fairness:

> No thread can be blocked indefinitely on an `MVar` unless another thread holds that `MVar` indefinitely.

In other words, if a thread T is blocked in `takeMVar`, and there are regular `putMVar` operations on the same `MVar`, then it is guaranteed that at some point thread T's `takeMVar` will return. In GHC this guarantee is implemented by keeping blocked threads in a FIFO queue attached to the `MVar`, so eventually every thread in the queue will get to complete its operation as long as there are other threads performing regular `putMVar` operations (an equivalent guarantee applies to threads blocked in `putMVar` when there are regular `takeMVar`s). Note that it is not enough to merely *wake up* the blocked thread, because another thread might run first and take (respectively put) the `MVar`, causing the newly woken thread to go to the back of the queue again, which would invalidate the fairness guarantee. The implementation must therefore atomically wake up the blocked thread *and* perform the blocked operation, which is exactly what GHC does.

Fairness in practice Recall our example from Section 8, where we had two threads, one printing `A`s and the other printing `B`s, and the output was often perfect alternation between the two: `ABABABABABABAB`. This is an example of the fairness guarantee in practice. The `stdout` handle is represented by an `MVar`, so when both threads attempt to call `takeMVar` to operate on the handle, one of them wins and the other becomes blocked. When the winning thread completes its operation and calls `putMVar`, the scheduler wakes up the blocked thread *and* completes its blocked `takeMVar`, so the original winning thread will immediately block when it tries to re-acquire the handle. Hence this leads to perfect alternation between the two threads. The only way that the alternation pattern can be broken is if one thread is pre-empted while it is not holding the `MVar`; indeed this does happen from time to time, as we see the occasional long string of a single letter in the output.

A consequence of the fairness implementation is that, when multiple threads are blocked, *we only need to wake up a single thread.* This single wakeup property is a particularly important performance characteristic

when a large number of threads are contending for a single `MVar`. As we shall see later, it is the fairness guarantee together with the single-wakeup property which means that `MVar`s are not completely subsumed by Software Transactional Memory.

10 Cancellation: Asynchronous Exceptions

In an interactive application, it is often important for one thread to be able to *interrupt* the execution of another thread when some particular condition occurs. Some examples of this kind of behaviour in practice include:

- In a web browser, the thread downloading the web page and the thread rendering the page need to be interrupted when the user presses the "stop" button.
- A server application typically wants to give a client a set amount of time to issue a request before closing its connection, so as to avoid dormant connections using up resources.
- An application in which a compute-intensive thread is working (say, rendering a visualisation of some data), and the input data changes due to some user input.

The crucial design decision in supporting cancellation is whether the intended victim should have to poll for the cancellation condition, or whether the thread is immediately cancelled in some way. This is a tradeoff:

1. If the thread has to poll, there is a danger that the programmer may forget to poll regularly enough, and the thread will become unresponsive, perhaps permanently so. Unresponsive threads lead to hangs and deadlocks, which are particularly unpleasant from a user's perspective.
2. If cancellation happens asynchronously, critical sections that modify state need to be protected from cancellation, otherwise cancellation may occur mid-update leaving some data in an inconsistent state.

In fact, the choice is really between doing only (1), or doing both (1) and (2), because if (2) is the default, protecting a critical section amounts to switching to polling behaviour for the duration of the critical section.

In most imperative languages it is unthinkable for (2) to be the default, because so much code is state-modifying. Haskell has a distinct advantage in this area, however: most code is purely functional, so it can be safely aborted or suspended, and later resumed, without affecting correctness. Moreover our hand is forced: purely functional code cannot by definition poll for the cancellation condition, so it must be cancellable by default.

Therefore, fully-asynchronous cancellation is the only sensible default in Haskell, and the design problem reduces to deciding how cancellation appears to code in the `IO` monad.

It makes sense for cancellation to behave like an exception, since exceptions are already a fact of life in the `IO` monad, and the usual idioms for writing `IO` monad code include exception handlers to release resources and clean up in the event of an error. For example, to perform an operation that requires a temporary file, we would use the `bracket` combinator to ensure that the temporary file is always removed, even if the operation raises an exception:

```
  bracket (newTempFile "temp")
          (\file -> removeFile file)
          (\file -> ...)
```

where `bracket` is defined thus:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after during = do
  a <- before
  c <- during a `onException` after a
  after a
  return c
```

and `onException` executes its first argument, and if an exception is thrown, executes its second argument before re-throwing the exception.

```
onException :: IO a -> IO b -> IO a
```

We want exception handlers to run in the event of cancellation, so cancellation should be an exception. However, there's a fundamental difference between the kind of exception thrown by `openFile` when the file does not exist, for example, and an exception that may arise *at any time* because the user pressed the "stop" button. We call the latter kind an *asynchronous* exception, for obvious reasons. (We do not review the Haskell support for *synchronous* exceptions here; for that see the Haskell 2010 report [9] and the documentation for the `Control.Exception` module).

To initiate an asynchronous exception, Haskell provides the `throwTo` primitive which throws an exception from one thread to another [3]:

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

the `Exception` constraint requires that the exception value being thrown is an instance of the `Exception` class, which implements a simple hierarchy [4]. The `ThreadId` is a value previously returned by `forkIO`, and may refer to a thread in any state: running, blocked, or finished (in the latter case, `throwTo` is a no-op).

To illustrate the use of `throwTo`, we now elaborate the earlier example in which we downloaded several web pages concurrently, to allow the user to hit `'q'` at any time to stop the downloads.

First, we will extend our `Async` mini-API to allow cancellation. We add one operation:

```
cancel :: Async a -> IO ()
```

which cancels an existing `Async`. If the operation has already completed, `cancel` has no effect. The `wait` operation cannot just return the result of the `Async` any more, since it may have been cancelled. Therefore, we extend `wait` to return `Either SomeException a`, containing either the exception raised during the operation, or its result:

```
wait :: Async a -> IO (Either SomeException a)
```

(`SomeException` is the root of the exception hierarchy in Haskell.) In order to implement the new interface, we need to extend the `Async` type to include the `ThreadId` of the child thread, and the `MVar` holding the result must now hold `Either SomeException a`.

```
data Async a = Async ThreadId (MVar (Either SomeException a))
```

Given this, the implementation of `cancel` just throws an exception to the thread:

```
cancel :: Async a -> IO ()
cancel (Async t var) = throwTo t ThreadKilled
```

(`ThreadKilled` is an exception provided by the Haskell exception library and is typically used for cancelling threads in this way.) The implementation of `wait` is trivial. The remaining piece of the implementation is the `async` operation, which must now include an exception handler to catch the exception and store it in the `MVar`:

```
async :: IO a -> IO (Async a)
async io = do
   m <- newEmptyMVar
   t <- forkIO (do r <- try io; putMVar m r)
   return (Async t m)
```

where `try` is a function provided by the `Control.Exception` library:

```
try :: Exception e => IO a -> IO (Either e a)
```

Now, we can change the `main` function of the example to support cancelling the downloads:

```
1  main = do
2    as <- mapM (async.http) sites
3
4    forkIO $ do
5       hSetBuffering stdin NoBuffering
6       forever $ do
7          c <- getChar
8          when (c == 'q') $ mapM_ cancel as
9
10   rs <- mapM wait as
11   printf "%d/%d finished\n" (length (rights rs)) (length rs)
```

Line 2 starts the downloads as before. Lines 4–8 fork a new thread that repeatedly reads characters from the standard input, and if a `q` is found, calls `cancel` on all the `Async`s. Line 10 waits for all the results (complete or cancelled), and line 11 emits a summary with a count of how many of the operations completed without being cancelled. If we run the sample[23] and hit `‘q‘` fast enough, we see something like this:

```
downloaded: http://www.google.com (14538 bytes, 0.17s)
downloaded: http://www.bing.com (24740 bytes, 0.22s)
q2/5 finished
```

Note that this works even though the program is sitting atop a large and complicated HTTP library that provides no direct support for either cancellation or asynchronous I/O. Haskell's support for cancellation is modular in this respect: most library code needs to do nothing to support it, although there are some simple and unintrusive rules that need to be followed when dealing with state, as we shall see in the next section.

10.1 Masking asynchronous exceptions

As we mentioned earlier, the danger with fully asynchronous exceptions is that one might fire while we are in the middle of updating some shared state, leaving the data in an inconsistent state, and with a high probability leading to mayhem later.

Hence, we certainly need a way to control the delivery of asynchronous exceptions during critical sections. But we must tread carefully: it would be easy to provide the programmer with a way to turn off asynchronous exception delivery temporarily, but such a facility is in fact not what we really need.

Consider the following problem: a thread wishes to call `takeMVar`, perform an operation depending on the value of the `MVar`, and finally put the result of the operation in the `MVar`. The code must be responsive to asynchronous exceptions, but it should be safe: if an asynchronous exception arrives after the `takeMVar`, but before the final `putMVar`, the `MVar` should not be left empty, instead the original value should be replaced.

If we code up this problem using the facilities we already seen so far, we might end up with something like this:

```
1 problem m f = do
2   a <- takeMVar m
3   r <- f a `catch` \e -> do putMVar m a; throw e
4   putMVar m r
```

There are at least two points where, if an asynchronous exception strikes, the invariant will be violated. If an exception strikes between lines 2 and 3,

[23] full code is in the sample `geturlscancel.hs`

or between lines 3 and 4, the `MVar` will be left empty. In fact, there is no way to shuffle around the exception handlers to ensure the `MVar` is always left full. To fix this problem, Haskell provides the `mask` combinator[24]:

```
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

The type looks a bit confusing, but it isn't really[25]. The `mask` operation defers the delivery of asynchronous exceptions for the duration of its argument, and is used like this:

```
1 problem m f = mask $ \restore -> do
2   a <- takeMVar m
3   r <- restore (f a) `catch` \e -> do putMVar m a; throw e
4   putMVar m r
```

`mask` is applied to a *function*, that takes as its argument a function `restore`, that can be used to restore the delivery of asynchronous exceptions to its present state. If we imagine shading the entire argument to `mask` except for the expression `(f a)`, asynchronous exceptions cannot be raised in the shaded portions.

This solves the problem that we had previously, since now an exception can only be raised while `(f a)` is working, and we have an exception handler to catch any exceptions in that case. But a new problem has been introduced: `takeMVar` might block for a long time, but it is inside the `mask` and so the thread will be unresponsive for that time. Furthermore there's no good reason to mask exceptions during `takeMVar`; it would be safe for exceptions to be raised right up until the point where `takeMVar` returns. Hence, this is exactly the behaviour that Haskell defines for `takeMVar`: we designate a small number of operations, including `takeMVar`, as *interruptible*. Interruptible operations may receive asynchronous exceptions even inside `mask`.

What justifies this choice? Think of `mask` as "switching to polling mode" for asynchronous exceptions. Inside a `mask`, asynchronous exceptions are no longer asynchronous, but they can still be raised by certain operations. In other words, asynchronous exceptions become *synchronous* inside `mask`.

All operations which may block indefinitely[26] are designated as interruptible. This turns out to be the ideal behaviour in many situations, as in `problem` above.

In fact, we can provide higher level combinators to insulate programmers from the need to use `mask` directly. For example, the function `problem` above is generally useful when working with `MVar`s, and is provided under the name `modifyMVar_` in the `Control.Concurrent.MVar` library.

[24] Historical note: the original presentation of asynchronous exceptions used a pair of combinators `block` and `unblock` here, but `mask` was introduced in GHC 7.0.1 to replace them as it has a more modular behaviour.

[25] for simplicity here we are using a slightly less general version of `mask` than the real one in the `Control.Exception` library.

[26] except foreign calls, for technical reasons

10.2 Asynchronous-exception safety

All that is necessary for most code to be safe in the presence of asynchronous exceptions is to use operations like `modifyMVar_` instead of `takeMVar` and `putMVar` directly. For example, consider the buffered channels that we defined earlier. As defined, the operations are not asynchronous-exception-safe; for example, `writeChan` was defined like this:

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar writeVar
  putMVar writeVar new_hole
  putMVar old_hole (Item val new_hole)
```

there are several windows here where if an asynchronous exception occurs, an `MVar` will be left empty, and subsequent users of the `Chan` will deadlock. To make it safe, we use `modifyMVar_`:

```
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  modifyMVar_ writeVar $ \old_hole -> do
    putMVar old_hole (Item val new_hole)
    return new_hole
```

We saw a use of the `bracket` function earlier; in fact, `bracket` is defined with `mask` in order to make it asynchronous-exception-safe:

```
bracket before after thing =
  mask $ \restore -> do
    a <- before
    r <- restore (thing a) `onException` after a
    _ <- after a
    return r
```

10.3 Timeouts

A good illustration of programming with asynchronous exceptions is to write a function that can impose a time limit on a given action. We want to provide the timeout wrapper as a combinator of the following type:

```
timeout :: Integer -> IO a -> IO (Maybe a)
```

where `timeout` t m has the following behaviour:

1. `timeout` t m behaves exactly like `fmap Just` m if m returns a result or raises an exception (including an asynchronous exception), within t microseconds.

2. otherwise, m is sent an asynchronous exception of the form `Timeout` u. `Timeout` is a new datatype that we define, and u is a unique value of type `Unique`, distinguishing this particular instance of `timeout` from any other. The call to `timeout` then returns `Nothing`.

Listing 1: implementation of `timeout`

```
1  timeout n m
2      | n <  0    = fmap Just m
3      | n == 0    = return Nothing
4      | otherwise = do
5          pid <- myThreadId
6          u <- newUnique
7          let ex = Timeout u
8          handleJust
9              (\e -> if e == ex then Just () else Nothing)
10             (\_ -> return Nothing)
11             (bracket (forkIO $ do threadDelay n
12                                   throwTo pid ex)
13                      (\t -> throwTo t ThreadKilled)
14                      (\_ -> fmap Just m))
```

The implementation is not expected to implement real-time semantics, so in practice the timeout will only be approximately t microseconds. Note that (1) requires that m is executed in the context of the current thread, since m could call `myThreadId`, for example. Also, another thread throwing an exception to the current thread with `throwTo` will expect to interrupt m.

The code for `timeout` is shown in Listing 1; this implementation was taken from the library `System.Timeout` (with some cosmetic changes for presentation here). The implementation is tricky to get right. The basic idea is to fork a new thread that will wait for t microseconds and then call `throwTo` to throw the `Timeout` exception back to the original thread; that much seems straightforward enough. However, we must ensure that this thread cannot throw its `Timeout` exception after the call to `timeout` has returned, otherwise the `Timeout` exception will leak out of the call, so `timeout` must kill the thread before returning.

Here is how the implementation works, line by line:

1–2 Handle the easy cases, where the timeout is negative or zero.

5 find the `ThreadId` of the current thread

6–7 make a new `Timeout` exception, by generating a unique value with `newUnique`

8-14 `handleJust` is an exception handler, with the following type:

```
handleJust :: Exception e
           => (e -> Maybe b) -> (b -> IO a) -> IO a
           -> IO a
```

Its first argument (line 9) selects which exceptions to catch: in this case, just the `Timeout` exception we defined on line 7. The second

argument (line 10) is the exception handler, which in this case just returns `Nothing`, since timeout occurred.

Lines 11–14 are the computation to run in the exception handler. `bracket` (Section 10) is used here in order to fork the child thread, and ensure that it is killed before returning.

11-12 fork the child thread. In the child thread we wait for n microseconds with `threadDelay`, and then throw the `Timeout` exception to the parent thread with `throwTo`.

13 always kill the child thread before returning.

14 the body of `bracket`: run the computation `m` passed in as the second argument to `timeout`, and wrap the result in `Just`.

The reader is encouraged to verify that the implementation works by thinking through the two cases: either `m` completes and returns `Just x` at line 14, or, the child thread throws its exception while `m` is still working.

There is one tricky case to consider: what happens if *both* the child thread and the parent thread try to call `throwTo` at the same time (lines 12 and 13 respectively)? Who wins?

The answer depends on the semantics of `throwTo`. In order for this implementation of `timeout` to work properly, it must not be possible for the call to `bracket` at line 11 to return while the `Timeout` exception can still be thrown, otherwise the exception can leak. Hence, the call to `throwTo` that kills the child thread at line 13 must be synchronous: once this call returns, the child thread cannot throw its exception any more. Indeed, this guarantee is provided by the semantics of `throwTo`: a call to `throwTo` only returns after the exception has been raised in the target thread[27]. Hence, `throwTo` may block if the child thread is currently masking asynchronous exceptions with `mask`, and because `throwTo` may block, it is therefore *interruptible* and may itself receive asynchronous exceptions.

Returning to our "who wins" question above, the answer is "exactly one of them", and that is precisely what we require to ensure the correct behaviour of `timeout`.

10.4 Asynchronous exceptions: reflections

Abstractions like `timeout` are certainly difficult to get right, but fortunately they only have to be written once. We find that in practice dealing with asynchronous exceptions is fairly straightforward, following a few simple rules:

- Use `bracket` when acquiring resources that need to be released again.

[27]Note: a different semantics was originally described in Marlow et al. [3].

- Rather than `takeMVar` and `putMVar`, use `modifyMVar_` (and friends) which have built-in asynchronous exception safety.
- If state handling starts getting complicated with multiple layers of exception handlers, then there are two approaches to simplifying things:
  - Switching to polling mode with `mask` can help manage complexity. The GHC I/O library, for example, runs entirely inside `mask`. Note that inside `mask` it is important to remember that asynchronous exceptions can still arise out of interruptible operations; the documentation contains a list of operations that are guaranteed *not* to be interruptible.
  - Using Software Transactional Memory (STM) instead of `MVar`s or other state representations can sweep away all the complexity in one go. We will describe STM in Section 11.

The rules are usually not onerous: remember this only applies to code in the `IO` monad, so the vast swathes of purely-functional library code available for Haskell is all safe by construction. We find that most `IO` monad code is straightforward to make safe, and if things get complicated falling back to either `mask` or STM is a satisfactory solution.

In exchange for following the rules, however, Haskell's approach to asynchronous exceptions confers many benefits.

- Many exceptional conditions map naturally onto asynchronous exceptions. For example, stack overflow and user interrupt (e.g. control-C at the console) are mapped to asynchronous exceptions in Haskell. Hence, control-C not only aborts the program but does so cleanly, running all the exception handlers. Haskell programmers have to do nothing to enable this behaviour.
- Constructs like `timeout` always work, even with third-party library code.
- Threads never just die in Haskell, it is guaranteed that a thread always gets a chance to clean up and run its exception handlers.

11 Software Transactional Memory

Software Transactional Memory (STM) is a technique for simplifying concurrent programming by allowing multiple state-changing operations to be grouped together and performed as a single atomic operation. Strictly speaking, "Software Transactional Memory" is an implementation technique, whereas the language construct we are interested in is "atomic blocks".

Listing 2: the interface provided by `Control.Concurrent.STM`

```
data STM a -- abstract
instance Monad STM -- amongst other things

atomically :: STM a -> IO a

data TVar a -- abstract
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

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

throwSTM  :: Exception e => e -> STM a
catchSTM  :: Exception e => STM a -> (e -> STM a) -> STM a
```

Unfortunately the former term has stuck, and so the language-level facility is called STM.

STM solves a number of problems that arise with conventional concurrency abstractions, that we describe here through a series of examples. For reference throughout the following section, the types and operations of the STM interface are collected in Listing 2.

Imagine the following scenario: a window manager that manages multiple desktops. The user may move windows from one desktop to another, while at the same time, a program may request that its own window moves from its current desktop to another desktop. The window manager uses multiple threads: one to listen for input from the user, one for each existing window to listen for requests from those programs, and one thread that renders the display to the user.

How should the program represent the state of the display? One option is to put it all in a single `MVar`:

```
type Display = MVar (Map Desktop (Set Window))
```

and this would work, but the `MVar` is a single point of contention. For example, the rendering thread, which only needs to look at the currently displayed desktop, could be blocked by a window on another desktop moving itself.

So perhaps we can try to allow more concurrency by having a separate `MVar` for each desktop:

```
type Display = Map Desktop (MVar (Set Window))
```

unfortunately this approach quickly runs into problems. Consider an operation to move a window from one desktop to another:

```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = do
```

```
  wa <- takeMVar ma
  wb <- takeMVar mb
  putMVar ma (Set.delete win wa)
  putMVar mb (Set.insert win wb)
 where
  ma = fromJust (Map.lookup disp a)
  mb = fromJust (Map.lookup disp b)
```

Note that we must take both `MVar`s before we can put the results: otherwise another thread could potentially observe the display in a state in which the window we are moving does not exist. But this raises a problem: what if there is concurrent call to `moveWindow` trying to move a window in the opposite direction? Both calls would succeed at the first `takeMVar`, but block on the second, and the result is a deadlock. This is an instance of the classic Dining Philosophers problem.

One solution is to impose an ordering on the `MVar`s, and require that all agents take `MVar`s in the correct order and release them in the opposite order. That is inconvenient and error-prone though, and furthermore we have to extend our ordering to any other state that we might need to access concurrently. Large systems with many locks (e.g. Operating Systems) are often plagued by this problem, and managing the complexity requires building elaborate infrastructure to detect ordering violations.

Transactional memory provides a way to avoid this deadlock problem without imposing a requirement for ordering on the programmer. To solve the problem using STM, we replace `MVar` with `TVar`:

```
type Display = Map Desktop (TVar (Set Window))
```

`TVar` stands for "transactional variable", and it is a mutable variable that can only be read or written within a transaction. To implement `moveWindow`, we simply perform the necessary operations on `TVar`s in the STM monad, and wrap the whole sequence in `atomically`:

```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = atomically $ do
  wa <- readTVar ma
  wb <- readTVar mb
  writeTVar ma (Set.delete win wa)
  writeTVar mb (Set.insert win wb)
 where
  ma = fromJust (Map.lookup a disp)
  mb = fromJust (Map.lookup b disp)
```

The code is almost identical to the `MVar` version, but the behaviour is quite different: the sequence of operations inside `atomically` happens indivisibly as far as the rest of the program is concerned. No other thread can observe an intermediate state; the operation has either completed, or it has not started yet. What's more, there is no requirement that we read both `TVar`s before we write them, this would be fine too:

```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = atomically $ do
  wa <- readTVar ma
  writeTVar ma (Set.delete win wa)
  wb <- readTVar mb
  writeTVar mb (Set.insert win wb)
 where
  ma = fromJust (Map.lookup disp a)
  mb = fromJust (Map.lookup disp b)
```

So STM is far less error-prone here. The approach also scales to any number of `TVar`s, so we could easily write an operation that moves the windows from all other desktops to the current desktop, for example.

Now suppose that we want to swap two windows, moving window W from desktop A to B, and simultaneously V from B to A. With the `MVar` representation we would have to write a special-purpose operation to do this, because it has to take the `MVar`s for A and B (in the right order), and then put both `MVar`s back with the new contents. With STM, however, we can express this much more neatly as a composition. First we need to expose a version of `moveWindow` without the `atomically` wrapper:

```
moveWindowSTM :: Display -> Window -> Desktop -> Desktop
              -> STM ()
moveWindowSTM disp win a b = do ...
```

and then we can define `swapWindows` by composing two `moveWindowSTM` calls:

```
swapWindows :: Display
            -> Window -> Desktop
            -> Window -> Desktop
            -> IO ()
swapWindows disp w a v b = atomically $ do
  moveWindowSTM disp w a b
  moveWindowSTM disp v b a
```

This demonstrates the *composability* of STM operations: any operation of type `STM a` can be composed with others to form a larger atomic transaction. For this reason, `STM` operations are usually provided without the `atomically` wrapper, so that clients can compose them as necessary, before finally wrapping the entire operation in `atomically`.

So far we have covered the basic facilities of STM, and shown that STM can be used to make atomicity scale in a composable way. STM confers a qualitative improvement in expressibility and robustness when writing concurrent programs. The benefits of STM in Haskell go further, however: in the following sections we show how STM can be used to make blocking abstractions compose, and how STM can be used to manage complexity in the presence of failure and interruption.

11.1 Blocking

An important part of concurrent programming is dealing with *blocking*; when we need to wait for some condition to be true, or to acquire a particular resource. STM provides an ingenious way to do this, with a single operation:

```
retry :: STM a
```

the meaning of `retry` is simply "run the current transaction again". That seems bizarre - why would we want to run the current transaction again? Well, for one thing, the contents of some `TVars` that we have read may have been changed by another thread, so re-running the transaction may yield different results. Indeed, there's no point re-running the transaction *unless* it is possible that something different might happen, and the runtime system knows this, so `retry` waits until a `TVar` that was read in the current transaction has been written to, and then triggers a re-run of the current transaction. Until that happens, the current thread is blocked.

ToDo: perhaps rearrange the text here, at least one reader was confused because the discussion of "not busy waiting" occurs before the concrete example.

As a concrete example, we can use `retry` to implement the rendering thread in our window-manager example. The behaviour we want is this:

- One desktop is designated as having the *focus*. The focussed desktop is the one displayed by the rendering thread.
- The user may request that the focus be changed at any time.
- Windows may move around and appear or disappear of their own accord, and the rendering thread must update its display accordingly.

We are supplied with a function `render` which handles the business of rendering windows on the display. It should be called whenever the window layout changes[28]:

```
render :: Set Window -> IO ()
```

The currently focussed desktop is a piece of state that is shared by the rendering thread and some other thread that handles user input. Therefore we represent that by a `TVar`:

```
type UserFocus = TVar Desktop
```

Next, we define an auxiliary function `getWindows` that takes the `Display` and the `UserFocus`, and returns the set of windows to render, in the `STM` monad. The implementation is straightforward: read the current focus, and look up the contents of the appropriate desktop in the `Display`:

[28] we are assuming that the actual window contents are rendered via some separate means, e.g. compositing

```
getWindows :: Display -> UserFocus -> STM (Set Window)
getWindows disp focus = do
  desktop <- readTVar focus
  readTVar (fromJust (Map.lookup desktop disp))
```

Finally, we can implement the rendering thread. The general plan is to repeatedly read the current state with `getWindows` and call `render` to render it, but use `retry` to avoid calling `render` when nothing has changed. Here is the code:

```
1  renderThread :: Display -> UserFocus -> IO ()
2  renderThread disp focus = do
3    wins <- atomically $ getWindows disp focus
4    loop wins
5   where
6    loop wins = do
7      render wins
8      next <- atomically $ do
9                  wins' <- getWindows disp focus
10                 if (wins == wins')
11                     then retry
12                     else return wins'
13     loop next
```

First we read the current set of windows to display (line 3) and use this as the initial value for the `loop` (line 4). Lines 6-13 implement the loop. Each iteration calls `render` to display the current state (line 7), and then enters a transaction to read the next state. Inside the transaction we read the current state (line 9), and compare it to the state we just rendered (line 10); if the states are the same, there is no need to do anything, so we call `retry`. If the states are different, then we return the new state, and the loop iterates with the new state (line 13).

The effect of the `retry` is precisely what we need: it waits until the value read by `getWindows` could possibly be different, because another thread has successfully completed a transaction that writes to one of the `TVar`s that is read by `getWindows`. That encompasses both changes to the `focus` (because the user switched to a different desktop), and changes to the contents of the current desktop (because a window moved, appeared, or disappeared). Furthermore, changes to other desktops can take place without the rendering thread being woken up.

If it weren't for STM's `retry` operation, we would have to implement this complex logic ourselves, including implementing the signals between threads that modify the state and the rendering thread. This is anti-modular, because operations that modify the state have to know about the observers that need to act on changes. Furthermore, it gives rise to a common source of concurrency bugs: *lost wakeups*. If we forgot to signal the rendering thread, then the display would not be updated. In this case the effects are somewhat benign, but in a more complex scenario lost wakeups often lead

to deadlocks, because the woken thread was supposed to complete some operation on which other threads are waiting.

11.2 Implementing `MVar` with STM

ToDo: Finish this section.

```
newtype TMVar a = TMVar (TVar (Maybe a))

newTMVar :: a -> STM (TMVar a)
newTMVar a = do
  t <- newTVar (Just a)
  return (TMVar t)

newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = do
  t <- newTVar Nothing
  return (TMVar t)

takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar t) = do
  m <- readTVar t
  case m of
    Nothing -> retry
    Just a  -> do writeTVar t Nothing; return a

putTMVar :: TMVar a -> a -> STM ()
putTMVar (TMVar t) a = do
  m <- readTVar t
  case m of
    Nothing -> do writeTVar t (Just a); return ()
    Just _  -> retry
```

Why might one want to use `MVar` rather than `TMVar`?

- Performance: `MVar` is a bit faster than `TMVar`, due to the overheads of STM transactions.
- Fairness: recall from Section 9.2 that threads blocked on an `MVar` will be unblocked one at a time in FIFO order. In STM the situation is different: when a thread is blocked in `retry`, changes to any of the `TVars` it read during the transaction cause it to retry the transaction. If multiple threads are blocked on the same `TVar`, then the runtime cannot guarantee that waking up just one of them will be sufficient, because that would require knowledge of the nature of the transaction to be retried. So when a `TVar` is written, all threads blocked in `retry` after reading that `TVar` will be woken. Hence `TMVar` does not have the fairness or single-wakeup properties of `MVar`.

11.3 Async revisited: waiting for multiple `Asyncs`

ToDo: Finish this section.

```
data Async a = Async ThreadId (TMVar (Either SomeException a))

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyTMVarIO
  t <- forkIO (do r <- try action; atomically (putTMVar var r))
  return (Async t var)

wait :: Async a -> IO (Either SomeException a)
wait = atomically . waitSTM

waitSTM :: Async a -> STM (Either SomeException a)
waitSTM (Async _ var) = readTMVar var

cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled
```

ToDo: Note: not using forkFinally here, we'll introduce that later (or maybe sooner? it would go nicely in the section on asynchronous exceptions...)

```
waitAny :: [Async a] -> IO ()
waitAny asyncs =
  atomically $
    foldr1 orElse $
      map (void . waitSTM) asyncs
```

Now we can implement a version of our URL downloader that waits for the *first* URL to complete downloading and then stops[29]:

```
main = do
  as <- mapM (async.http) sites

  waitAny as
  mapM_ cancel as
  rs <- mapM wait as
  printf "%d/%d finished\n" (length (rights rs)) (length rs)
 where
   http url = do
     (page, time) <- timeit $ getURL url
     printf "downloaded: %s (%d bytes, %.2fs)\n" url (B.length
         page) time
```

When we run this, the output will be similar to the following:

```
downloaded: http://www.google.com (11448 bytes, 0.08s)
1/5 finished
```

ToDo: further things to add: we could do this with MVars by forking new threads to do the multiplexing, but it's ugly. Furthermore, waitSTM can be composed with other blocking operations as required.

[29] full sample code in `geturlsfirst.hs`

```
waitEither :: Async a -> Async b
           -> IO (Either (Either SomeException a)
                         (Either SomeException b))
waitEither left right =
  atomically $
    (Left  <$> waitSTM left)
      `orElse`
    (Right <$> waitSTM right)
```

11.4 Implementing channels with STM

For our fourth example of STM, we shall implement the `Chan` type from Section 9.1 using STM. As we'll see, using STM to implement `Chan` is rather less tricky than using `MVar`s, and furthermore we are able to add some more complex operations that were hard or impossible using `MVar`s.

The STM version of `Chan` is called `TChan`[30], and the interface we wish to implement is as follows:

```
data TChan a

newTChan   :: STM (TChan a)
writeTChan :: TChan a -> a -> STM ()
readTChan  :: TChan a -> STM a
```

that is, exactly the same as `Chan`, except that we renamed `Chan` to `TChan`. The full code for the implementation is given in Listing 3. The implementation is similar in structure to the `MVar` version in Section 9.1, so we do not describe it line by line, however we shall point out a few important details:

- All the operations are in the `STM` monad, so to use them they need to be wrapped in `atomically` (but they can also be composed, more about that later).
- Blocking in `readTChan` is implemented by the call to `retry` (line 19).
- Nowhere did we have to worry about what happens when a read executes concurrently with a write, because all the operations are atomic.

Something worth noting, although this is not a direct result of STM, is that the straightforward implementation of `dupChan` does not suffer from the problem that we had in Section 9.1, because `readTChan` does not remove elements from the list.

We now describe three distinct benefits of the STM implementation compared to using `MVar`s.

[30] the implementation is available in the module `Control.Concurrent.STM.TChan` from the `stm` package.

Listing 3: implementation of `TChan`

```
data TChan a = TChan (TVar (TVarList a))
                     (TVar (TVarList a))

type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)

newTChan :: STM (TChan a)
newTChan = do
  hole <- newTVar TNil
  read <- newTVar hole
  write <- newTVar hole
  return (TChan read write)

readTChan :: TChan a -> STM a
readTChan (TChan readVar _) = do
  listhead <- readTVar readVar
  head <- readTVar listhead
  case head of
    TNil -> retry
    TCons val tail -> do
        writeTVar readVar tail
        return val

writeTChan :: TChan a -> a -> STM ()
writeTChan (TChan _ writeVar) a = do
  new_listend <- newTVar TNil
  listend <- readTVar writeVar
  writeTVar writeVar new_listend
  writeTVar listend (TCons a new_listend)
```

More operations are possible. In Section 9.1 we mentioned the operation `unGetChan`, which could not be implemented with the desired semantics using `MVar`s. Here is its implementation with STM:

```
unGetTChan :: TChan a -> a -> STM ()
unGetTChan (TChan read _write) a = do
   listhead <- readTVar read
   newhead <- newTVar (TCons a listhead)
   writeTVar read newhead
```

The obvious implementation does the right thing here. Other operations that were not possible with `MVar`s are straightforward with STM, for example `isEmptyTChan`, the `MVar` version of which suffers from the same problem as `unGetChan`:

```
isEmptyTChan :: TChan a -> STM Bool
isEmptyTChan (TChan read _write) = do
  listhead <- readTVar read
  head <- readTVar listhead
  case head of
    TNil -> return True
    TCons _ _ -> return False
```

Composition of blocking operations. Suppose we wish to implement an operation `readEitherTChan` that can read an element from either of two channels. If both channels are empty it blocks; if one channel is non-empty it reads the value from that channel, and if both channels are non-empty it is allowed to choose which channel to read from. Its type is

```
readEitherTChan :: TChan a -> TChan b -> STM (Either a b)
```

We cannot implement this function with the operations introduced so far, but STM provides one more crucial operation that allows blocking transactions to be composed. The operation is `orElse`:

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

The operation `orElse a b` has the following behaviour:

- First `a` is executed. If `a` returns a result, then that result is immediately returned by the `orElse` call.
- If `a` instead called `retry`, then `a`*'s effects are discarded*, and `b` is executed instead.

We can use `orElse` to compose blocking operations atomically. Returning to our example, `readEitherTChan` could be implemented as follows:

```
readEitherTChan :: TChan a -> TChan b -> STM (Either a b)
readEitherTChan a b =
  fmap Left (readTChan a)
    `orElse`
  fmap Right (readTChan b)
```

This is a straightforward composition of the two `readTChan` calls, the only complication is arranging to tag the result with either `Left` or `Right` depending on which branch succeeds.

In the `MVar` implementation of `Chan` there is no way to implement the operation `readEitherChan` without elaborating the representation of `Chan` to support the synchronisation protocol that would be required (more discussion on implementing choice with `MVar`s can be found in Peyton Jones et al. [11]).

One thing to note is that `orElse` is left-biased; if both `TChan`s are non-empty, then `readEitherChan` will always return an element from the first one. Whether this is problematic or not depends on the application: something to be aware of is that the left-biased nature of `orElse` can have implications for fairness in some situations.

Asynchronous exception safety. Up until now we have said nothing about how exceptions in STM behave. The `STM` monad supports exceptions much like the `IO` monad, with two operations:

```
throwSTM  :: Exception e => e -> STM a
catchSTM  :: Exception e => STM a -> (e -> STM a) -> STM a
```

`throwSTM` throws an exception, and `catchSTM` catches exceptions and invokes a handler, just like `catch` in the `IO` monad. However, exceptions in STM are different in one vital way:

- In `catchSTM m h`, if `m` raises an exception, then *all of its effects are discarded*, and then the handler `h` is invoked. As a degenerate case, if there is no enclosing `catchSTM` at all, then all of the effects of the transaction are discarded and the exception is propagated out of `atomically`.

This behaviour of `catchSTM` was introduced in a subsequent amendment of Harris et al. [2]; the original behaviour in which effects were not discarded being generally regarded as much less useful. An example helps to demonstrate the motivation:

```
readCheck :: TChan a -> STM a
readCheck chan = do
  a <- readTChan chan
  checkValue a
```

`checkValue` imposes some extra constraints on the value read from the channel. However, suppose `checkValue` raises an exception (perhaps accidentally, e.g. divide-by-zero). We would prefer it if the `readTChan` had not happened, since an element of the channel would be lost. Furthermore, we would like `readCheck` to have this behaviour regardless of whether there is an enclosing exception handler or not. Hence `catchSTM` discards the effects of its first argument in the event of an exception.

The discarding-effects behaviour is even more useful in the case of *asynchronous* exceptions. If an asynchronous exception occurs during an STM transaction, the entire transaction is aborted (unless the exception is caught and handled, but handling asynchronous exceptions in STM is not something we typically want to do). So in most cases, asynchronous exception safety in STM consists of doing *absolutely nothing at all*. There are no locks to replace, so no need for exception handlers or `bracket`, and no need to worry about which critical sections to protect with `mask`.

The implementation of `TChan` given earlier is entirely safe with respect to asynchronous exceptions as it stands, and moreover any compositions of these operations are also safe.

STM provides a nice way to write code that is automatically safe with respect to asynchronous exceptions, so it can be useful even for state that is not shared between threads. The only catch is that we have to use STM consistently for all our state, but having made that leap, asynchronous exception safety comes for free.

11.5 Performance

As with most abstractions, STM has a runtime cost. If we understand the cost model, then we can avoid writing code that hits the bad cases. So in this section we give an informal description of the implementation of STM (at least in GHC), with enough detail that the reader can understand the cost model.

An STM transaction works by accumulating a *log* of `readTVar` and `writeTVar` operations that have happened so far during the transaction. The log is used in three ways:

- By storing `writeTVar` operations in the log rather than applying them to main memory immediately, discarding the effects of a transaction is easy; we just throw away the log. Hence, aborting a transaction has a fixed small cost.

- Each `readTVar` must traverse the log to check whether the `TVar` was written by an earlier `writeTVar`. Hence, `readTVar` is an $O(n)$ operation in the length of the log.

- Because the log contains a record of all the `readTVar` operations, it can be used to discover the full set of `TVar`s read during the transaction, which we need to know in order to implement `retry`.

When a transaction reaches the end, the STM implementation compares the log against the contents of memory using a two-phase locking protocol (details in Harris et al. [2]). If the current contents of memory matches the values read by `readTVar`, the effects of the transaction are *committed* to

memory atomically, and if not, the log is discarded and the transaction runs again from the beginning. The STM implementation in GHC does not use global locks; only the `TVar`s involved in the transaction are locked during commit, so transactions operating on disjoint sets of `TVar`s can proceed without interference.

The general rule of thumb when using STM is never to read an unbounded number of `TVar`s in a single transaction, because the $O(n)$ performance of `readTVar` then gives $O(n^2)$ for the whole transaction. Furthermore, long transactions are much more likely to fail to commit, because another transaction will probably have modified one or more of the same `TVar`s in the meantime, so there is a high probability of re-execution.

It is possible that a future STM implementation may use a different data structure to store the log, reducing the `readTVar` overhead to $O(\log\ n)$ or better (on average), but the likelihood that a long transaction will fail to commit would still be an issue. To avoid that problem intelligent contention-management is required, which is an area of active research.

ToDo: describe the implementation of retry

11.6 Summary

To summarise, STM provides several benefits for concurrent programming:

- **Composable atomicity**. We may construct arbitrarily large atomic operations on shared state, which can simplify the implementation of concurrent data structures with fine-grained locking.
- **Composable blocking**. We can build operations that make a choice between multiple blocking operations; something which is very difficult with `MVar`s and other low-level concurrency abstractions.
- **Robustness in the presence of failure and cancellation**. A transaction in progress is aborted if an exception occurs, so STM makes it easy to maintain invariants on state in the presence of exceptions.

11.7 Further reading

To find out more about STM in Haskell:

- Harris et al. [2], the original paper describing the design of Haskell's STM interface (be sure to get the revised version[31] which has the modified semantics for exceptions).
- "Beautiful Concurrency" a chapter in Wilson [15].

[31] `http://research.microsoft.com/people/simonpj/`

12 Higher-level concurrency abstractions

The preceding sections covered the basic interfaces for writing concurrent code in Haskell. These are enough for simple tasks, but for larger and more complex programs we need to raise the level of abstraction.

Earlier in Section 9 we introduced the `Async` interface for performing operations asynchronously and waiting for the results. In this section we will be revisiting that interface and expanding it with some more sophisticated functionality. In particular, we will

- provide a way to create an `Async` that is automatically cancelled if its parent dies, and
- provide a way to propagate exceptions from an `Async` to its parent.

What we are aiming for is the ability to build *trees of threads*, such that when a process dies for whatever reason, two things happen: any children it has are automatically terminated, and its parent is informed. This is part of the "let it crash" philosophy promoted by the designers of Erlang: the idea is based on the observation that getting error-recovery right is hard, and so rather than trying to build elaborate error-recovery machinery, we should simply program the normal case only and let every abnormal situation cause the thread (or process in the case of Erlang) to crash. The crash is typically caught by a supervisor process that knows how to restart the system into a known good state.

In order to implement the "let it crash" philosophy, we need to be able to spawn threads that will inform their parent when they go wrong. In Erlang this is done by *linking* processes together, a mechanism that is provided natively by the Erlang runtime. In Haskell, we can build equivalent functionality using the mechanisms introduced earlier: threads, asynchronous exceptions, and STM.

Once again we return to the `Async` API.

13 Shared concurrent data structures

ToDo: Discuss the trade-offs between IORef, MVar and STM.

14 High-speed concurrent server applications

Server-type applications that communicate with many clients simultaneously demand both a high degree of concurrency and high performance from the I/O subsystem. A good web server should be able to handle hundreds of thousands of concurrent connections, and service tens of thousands of requests per second.

Ideally, we would like to write these kinds of applications using threads. A thread is the right abstraction: it allows the developer to focus on programming the interaction with a single client, and then to lift this interaction to multiple clients by simply forking many instances of the single-client interaction in separate threads. To illustrate this idea we will describe a simple network server[32], with the following behaviour:

- The server accepts connections from clients on port 44444.
- If a client sends an integer n, the service responds with the value of $2n$
- If a client sends the string `"end"`, the server closes the connection.

First, we program the interaction with a single client. The function `talk` defined below takes a `Handle` for communicating with the client. The `Handle` is typically bound to a network socket, so data sent by the client can be read from the `Handle`, and data written to the `Handle` will be sent to the client.

```
1  talk :: Handle -> IO ()
2  talk h = do
3    hSetBuffering h LineBuffering
4    loop
5   where
6    loop = do
7      line <- hGetLine h
8      if line == "end"
9         then hPutStrLn h ("Thank you for using the " ++
10                         "Haskell doubling service.")
11        else do hPutStrLn h (show (2 * (read line :: Integer)))
12                loop
```

Line 3 sets the buffering mode for the `Handle` to line-buffering; if we don't do that then output sent to the `Handle` will be buffered up by the I/O layer until there is a full block (which is more efficient for large transfers, but not useful for interactive applications). Then we enter a loop to respond to requests from the client. Each iteration of the loop reads a new line of text (line 7), and then checks whether the client sent `"end"`. If so, we emit a polite message and return (line 8). If not, we attempt to interpret the line as an integer and to write the value obtained by doubling it. Finally we call `loop` again to read the next request.

Having dealt with the interaction with a single client, we can now make this into a multi-client server using concurrency. The `main` function for our server is as follows:

```
1  main = withSocketsDo $ do
2    sock <- listenOn (PortNumber (fromIntegral port))
```

[32] the full code can be found in sample `server.hs`

```
3   printf "Listening on port %d\n" port
4   forever $ do
5      (handle, host, port) <- accept sock
6      printf "Accepted connection from %s: %s\n" host (show port
           )
7      forkIO (talk handle `finally` hClose handle)

9   port :: Int
10  port = 44444
```

On line 2 we create a network socket to listen on port 44444, and then we enter a loop to accept connections from clients (line 3). Line 5 accepts a new client connection: `accept` blocks until a connection request from a client arrives, and then returns a `Handle` for communicating with the client (here bound to `handle`) and some information about the client (here we bind `host` to the client's hostname). Line 6 reports the new connection, and on line 7 we call `forkIO` to create a new thread to handle the request. A little explanation is needed for the expression passed to `forkIO`:

```
talk handle `finally` hClose handle
```

`talk` is the single-client interaction that we defined above. The function `finally` is a standard exception-handling combinator. It is rather like a specialised version of `bracket`, and has the following type

```
finally :: IO a -> IO b -> IO a
```

with the behaviour that `a `finally` b` behaves exactly like `a`, except that `b` is always performed after `a` returns or throws an exception. Here we are using `finally` to ensure that the `Handle` for communicating with the client is always closed, even if `talk` throws an exception. If we didn't do this, the `Handle` would eventually be garbage collected, but in the meantime it would consume resources which might lead to the program failing due to lack of file descriptors. It is always a good idea to close `Handle`s when you're finished with them.

Having forked a thread to handle this client, the main thread then goes back to accepting more connections. All the active client connections and the main thread run concurrently with each other, so the fact that the server is handling multiple clients will be invisible to any individual client (unless the server becomes overloaded).

So, making our concurrent server was simple - we did not have to change the single-client code at all, and the code to lift it to a concurrent server was only a handful of lines. We can verify that it works: in one window we start the server

```
$ ./server
```

in another window we start a client, and try a single request[33]:

[33] `nc` is the netcat program, which is useful for simple network interaction

```
$ nc localhost 44444
22
44
```

Next we leave this client running, and start another client:

```
$ ghc -e 'mapM_ print [1..]' | nc localhost 44444
2
4
6
...
```

this client exercises the server a bit more by sending it a continuous stream of numbers to double. For fun, try starting a few of these. Meanwhile we can switch back to our first client, and observe that it is still being serviced:

```
$ nc localhost 44444
22
44
33
66
```

finally we can end the interaction with a client by typing `end`:

```
end
Thank you for using the Haskell doubling service.
```

This was just a simple example, but the same ideas underly several high-performance web-server implementations in Haskell. Furthermore, with no additional effort at all, the same server code can make use of multiple cores simply by compiling with `-threaded` and running with `+RTS -N`.

There are two technologies that make this structure feasible in Haskell:

- GHC's very lightweight threads mean that having one thread per client is practical.
- The IO manager [10] handles outstanding blocked I/O operations using efficient operating-system primitives (e.g. the `epoll` call in Unix), which allows us to have many thousands of threads doing I/O simultaneously with very little overhead.

Were it not for lightweight threads and the IO manager, we would have to resort to collapsing the structure into a single event loop (or worse, multiple event loops to take advantage of multiple cores). The event loops style loses the single-client abstraction, instead all clients have to be dealt with simultaneously, which can be complicated if there are different kinds of client

with different behaviours. Furthermore we have to represent the state of each client somehow, rather than just writing the straight-line code as we did in `talk` above. Imagine extending `talk` to implement a more elaborate protocol with several states — it would be reasonably straightforward with the single client abstraction, but representing each state and the transitions explicitly would quickly get complicated.

We have ignored many details that would be necessary in a real server application. The reader is encouraged to think about these and to try implementing any required changes on top of the provided sample code:

- What should happen if the user interrupts the server with control-C? (control-C is implemented as an asynchronous exception `Interrupted` which is sent to the main thread).
- What happens in `talk` if the line does not parse as a number?
- What happens if the client cuts the connection prematurely, or the network goes down?
- Should there be a limit on the number of clients we serve simultaneously?
- Can we log the activity of the server to a file?

14.1 A chat server

Following on from the simple server example in the previous section, we now consider a more realistic example: a network chat server. A chat server enables multiple clients to connect and type messages to each other interactively. Real chat servers (e.g. IRC) have multiple channels and allow clients to choose which channels to participate in; for simplicity we will be building a chat server that has a single channel, whereby every message is seen by every client.

The informal specification for the server is as follows:

- When a client connects, the server requests the name that the client will be using. The client must choose a name that is not currently in use, or the server will request another name.
- Each line received from the client is interpreted as a command, which is one of

`/tell` *name message* Sends *message* to the user *name*.

`/kick` *name* Disconnects user *name*[34].

[34] In real chat servers this command would typically only be available to privileged users, but for simplicity here we will allow any user to kick any other user.

`/quit` Disconnects the current client.

message.. Any other string (not beginning with `/`) is broadcast as a message to all the connected clients.

- Whenever a client connects or disconnects, all other connected clients are notified.
- We will be handling errors correctly, and aiming for consistent behaviour. For exmple, when two clients connect at the same time, one of them is always deemed to have connected first and gets notified about the other client connecting.
- If two clients simultaneously try to kick each other, only one of them will succeed. This may seem obvious, but as we shall see it is easy to get this wrong.

Architecture. As before, the basic architecture is to have a single server thread that accepts connections and forks a new thread for each client connection. However, what makes this example somewhat more interesting than the simple server of the previous section, is that a client thread must listen for events from multiple sources:

- It receives commands over the network,
- it receives messages from other connected clients, which must be sent back over the network,
- it can be kicked at any time by another client.

Now, we cannot structure the client as a single thread, because Concurrent Haskell does not provide a way to listen for both network traffic and local communication (be it an `MVar` or `STM`) at the same time. So we need two threads for each client, one that listens for network traffic, and one that listens for events from other clients.

Another constraint is that we should avoid sending data to the network socket from multiple threads, because they might get arbitrarily interleaved. We could add some locking to gain atomicity, but it is simpler to have just one thread writing to the socket per client.

So to summarise, we need

- One thread, the *receive* thread, that reads commands from the network socket,
- another thread that listens for messages from other clients and kick requests. We shall designate this thread the thread that sends information back to the client over the network, and hence call it the *send* thread.

To keep things simple, we will move as much logic as possible into the send thread, leaving the receive thread to just repeatedly read lines of text from the socket and forward them to the send thread. This raises the question of how the send thread should listen for the various events it needs to act upon: lines of text from the user, messages from other clients, and kick requests.

We might consider having the send thread read from a single `Chan`, and send requests both from the receive thread and other clients down this channel. However, this would make it difficult to implement our consistency requirement for the `/kick` command: if a `/kick` command resulted in a message being queued on a `Chan`, then it would be entirely possible for two threads to simultaneously kick each other, because the first `/kick` command would be in the `Chan` of the second client, meanwhile the second client could enqueue a `/kick` message on the `Chan` of the first client.

Therefore `/kick` should be given a high priority: a client should only process another message if there is no `/kick` pending for it. The easiest way to service multiple sources of events like this is with an STM transaction: we simply store a `/kick` request in one `TVar` and the rest of the messages in a separate `TChan`, and the send thread can check both on each iteration.

Indeed, if we use STM there is no need to use a single `TChan`; we could have one `TChan` for the receive thread to communicate with the send thread, and another `TChan` for the other clients to send messages on. However, using a single channel allows us to retain the ordering of all messages, which might make things more predictable for the user, hence we stick to the design of a single `TChan` for messages and a `TVar` for kick requests.

Client data. Now that we have established the main architectural design, we can fill in the details[35]. First, the data structure describing a client:

```
type ClientName = String

data Client = Client
  { clientName     :: ClientName
  , clientHandle   :: Handle
  , clientKicked   :: TVar (Maybe String)
  , clientSendChan :: TChan Message
  }
```

We have one `TVar` indicating whether this client has been kicked (`clientKicked`). If this `TVar` contains `Just s`, then `s` is a string describing the reason for the client being kicked. The `TChan clientSendChan` carries all the other messages that may be sent to a client. The objects that it contains are of the type `Message`:

```
data Message = Notice String
             | Tell ClientName String
```

[35] the full code can be found in sample `chat/Main.hs`

```
              | Broadcast ClientName String
              | Command String
```

where, respectively: `Notice` is a message from the server, `Tell` is a private message from another client, `Broadcast` is a public message from another client, and `Command` is a line of text received from the user (via the receive thread).

Next, we define a useful function for sending a `Message` to a given `Client`:

```
sendMessage :: Client -> Message -> STM ()
sendMessage Client{..} msg =
    writeTChan clientSendChan msg
```

Note that this function is in the `STM` monad, not `IO`. We will be using it inside some STM transactions later.

Server data. The data structure that stores the server state is just a `TVar` containing a mapping from client names to `Clients`:

```
data Server = Server
  { clients :: TVar (Map ClientName Client)
  }

newServer :: IO Server
newServer = do
  c <- newTVarIO Map.empty
  return Server { clients = c }
```

This state must be accessible from all the clients, because each client needs to be able to broadcast to all the others. Here is how we broadcast a `Message` to all the clients:

```
broadcast :: Server -> Message -> STM ()
broadcast Server{..} msg = do
    clientmap <- readTVar clients
    F.mapM_ (\client -> sendMessage client msg) clientmap
```

Now, we will work top-down and write the code of the server. The `main` function is almost identical to the one used for the simple server in Section 14:

```
main :: IO ()
main = withSocketsDo $ do
  server <- newServer
  sock <- listenOn (PortNumber (fromIntegral port))
  printf "Listening on port %d\n" port
  forever $ do
      (handle, host, port) <- accept sock
      printf "Accepted connection from %s: %s\n" host (show
          port)
      forkFinally (talk server handle)
                  (\_ -> hClose handle)
```

```
port :: Int
port = 44444
```

the only difference being that we create a new empty server state up front by calling `newServer`, and pass this to each new client as an argument to `talk`.

Setting up a new client. When a new client connects, we need to do the following tasks:

- Ask the client for a user name,
- if the user name already exists, then ask for another name,
- otherwise, create a new `Client` and insert it into the `Server` state, ensuring that the `Client` will be removed when it disconnects or any failure occurs.
- Notify all existing clients that the new client has connected,
- set up the threads to handle the client connection and start processing messages.

Note the main source of trickiness here: we need to add the new client to the `Server` if and only if the user supplies us with a user name that is not already connected, and at the same time we must arrange to remove the new client after disconnection or failure, without there being any possibility that a stale `Client` is left lying around, which would lead to a memory leak.

Let's start by definiing `checkAddClient`, which takes a user name and attempts to add a new client with that name to the state, returning `Nothing` if a client with that name already exists, or `Just client` if the addition was succesful. It also broadcasts the event to all the other connected clients:

```
checkAddClient :: Server -> ClientName -> Handle -> IO (Maybe
    Client)
checkAddClient server@Server{..} name handle = atomically $ do
    clientmap <- readTVar clients
    if Map.member name clientmap
       then return Nothing
       else do client <- newClient name handle
               modifyTVar' clients $ Map.insert name client
               broadcast server $ Notice $ name ++ " has
                   connected"
               return (Just client)
```

and we will need a corresponding `removeClient` that removes the client again:

```
removeClient :: Server -> ClientName -> IO ()
removeClient server@Server{..} name = atomically $ do
    modifyTVar' clients $ Map.delete name
    broadcast server $ Notice $ name ++ " has disconnected"
```

Now we can put the pieces together. Unfortunately we can't reach for the usual tool for these situations, namely `bracket`, because our "resource acquisition" (`checkAddClient`) is conditional. So we need to write the code out explicitly:

```
talk :: Server -> Handle -> IO ()
talk server@Server{..} handle = do
    hSetNewlineMode handle universalNewlineMode
        -- Swallow carriage returns sent by telnet clients
    hSetBuffering handle LineBuffering
    readName
  where
    readName = do
      hPutStrLn handle "What is your name?"
      name <- hGetLine handle
      if null name
         then readName
         else do
                ok <- checkAddClient server name handle
                case ok of
                  Nothing -> do
                     hPrintf handle
                        "The name %s is in use, please choose
                            another\n" name
                     readName
                  Just client ->
                     runClient server client
                       `finally` removeClient server name
```

This is *almost* right, but strictly speaking we should mask asynchronous exceptions to eliminate any possibilty that an exception is received just after `checkAddClient` but before `runClient`, which would leave a stale client in the state. This is what `bracket` would have done for us, but since we're rolling our own logic here, we have to do the exception safety too (for reference the definition of `bracket` is given in Section 10.2).

The fixed version of `readName` is as follows:

```
    readName = do
      hPutStrLn handle "What is your name?"
      name <- hGetLine handle
      if null name
         then readName
         else mask $ \restore -> do
                ok <- checkAddClient server name handle
                case ok of
                  Nothing -> restore $ do
                     hPrintf handle
                        "The name %s is in use, please choose
                            another\n" name
                     readName
                  Just client ->
                     restore (runClient server client)
                       `finally` removeClient server name
```

Running the client. Having initialised the client, created the `Cient` data structure and added it to the `Server` state, we now need to create the client threads themselves and start processing events. The main functionality of the client will be implemented in a function `runClient`:

```
runClient :: Server -> Client -> IO ()
```

where `runClient` only returns or throws an exception when the client is to be disconnected. Recall that we need two threads per client: a *receive* thread to read from the network socket, and a *send* thread to listen for messages from other clients and to send messages back over the network. Having two separate threads adds some complication: what if one of the threads exits or dies with an exception? It will be easier if we first build a robust abstraction that deals with such issues. We shall call the combinator `concurrently`:

```
concurrently :: IO () -> IO () -> IO ()
```

`concurrently` takes two `IO` actions as arguments, and as the name suggests, runs them in separate threads. If either action exits, either normally or by throwing an exception, then the other action is terminated with `killThread` and the call to `concurrently` returns.

The basic idea is straightforward: create an `MVar`, fork a thread to run each action and `finally` put the `MVar`, then in the original thread we take the `MVar` and `finally` kill both threads (killing the thread that has already exited is harmless). Unfortunately the combinator we would like to use (`bracket`) isn't quite enough again, because we need to be able to call `restore` in our forked threads, and `bracket` doesn't make available the `restore` function that it obtains from `mask`. Nevertheless, we can define `concurrently` by instantiating the definition of `bracket`, which was given in Section 10.2:

```
concurrently left right = do
    done <- newEmptyMVar
    mask $ \restore -> do
        let
            spawn x = forkIO $ restore x `finally` tryPutMVar
                done ()
            stop threads = mapM_ killThread threads
        --
        tids <- mapM spawn [left,right]
        restore (takeMVar done) `onException` stop tids
        stop tids
```

`concurrently` is one more tool in your bag; the more of these tools we have, the less likely it is that you have to worry about calling `mask` directly from your application code.

Now that we have `concurrently`, we can write the code for `runClient`:

```
runClient :: Server -> Client -> IO ()
runClient server@Server{..} client@Client{..}
 = race_ send receive
```

```
where
   send = join $ atomically $ do
       k <- readTVar clientKicked
       case k of
           Just reason -> return $
               hPutStrLn clientHandle $ "You have been kicked:
                    " ++ reason
           Nothing -> do
               msg <- readTChan clientSendChan
               return $ do
                   continue <- handleMessage server client msg
                   when continue $ send

   receive = do
      msg <- hGetLine clientHandle
      atomically $ sendMessage client $ Command msg
      receive
```

So `runClient` is just `concurrently` applied to the `send` and `receive` threads. In the `receive` thread we read one line at a time from the client's `Handle` and forward it to the send thread as a `Command` message.

In the `send` thread, we have a transaction that tests two pieces of state: first, the `clientKicked TVar`, to see whether this client has been kicked, and if not, we take the next message from `clientSendChan` and act upon it. Note that `send` is expressed as `join` applied to the STM transaction: the `join` function is from `Control.Monad` and has type

```
join :: Monad m => m (m a) -> m a
```

where here `m` is instantiated to `IO`. The STM transaction returns an `IO` action, which is run by `join`, and in most cases this `IO` action returned will recursively invoke `send`.

Acting on a message is handled by the `handleMessage` function, which is entirely straightforward:

```
handleMessage :: Server -> Client -> Message -> IO Bool
handleMessage server client@Client{..} message =
  case message of
     Notice msg         -> output $ "*** " ++ msg
     Tell name msg      -> output $ "*" ++ name ++ "*: " ++ msg
     Broadcast name msg -> output $ "<" ++ name ++ ">: " ++ msg
     Command msg ->
       case words msg of
           ["/kick", who] -> do
               join $ atomically $ kick server client who
               return True
           "/tell" : who : what -> do
               atomically $ tell server clientName who (unwords
                    what)
               return True
           ["/quit"] ->
               return False
           ('/':_):_ -> do
```

```
                hPutStrLn clientHandle $ "Unrecognised command:
                    " ++ msg
                return True
            _ -> do
                atomically $ broadcast server $ Broadcast
                    clientName msg
                return True
where
  output s = do hPutStrLn clientHandle s; return True
```

Note that the function returns a `Bool` to indicate whether the caller should continue to handle more messages (`True`) or exit (`False`).

Recap. We have now given all the code for the chat server: less than 250 lines total, which is not at all bad considering that we have implemented a complete and usable chat client. Moreover, with no changes the client will scale to many thousands of connections, and will make use of multiple CPUs if they are available.

We needed one new tool: `concurrently`, which should be generally useful (indeed it is one of a family of similar abstractions).

We used STM to gain some useful consistency properties. It would be possible to redesign the server without STM, but it would no doubt be more difficult: an ordinary `Chan` would not be enough, we would have to build another abstraction, and as we have seen, building new abstractions with `MVar` can be a tricky business. STM is a great tool for ensuring complex consistency properties in concurrent applications, and our experience so far has been that beyond a certain level of complexity, STM becomes a necessity.

Care should be taken with STM with respect to performance, though: take a look for a minute at the definition of `broadcast` above. It is an STM transaction that operates on an unbounded number of `TChan`s, and thus builds an unbounded transaction. We noted earlier (Section 11.5) that long transactions should be avoided because they cost $O(n^2)$, so `broadcast` should be reimplemented to avoid this. One alternative would be to create a single broadcast channel and use `dupTChan` so that it can be read by all clients separately.

15 Concurrency and the Foreign Function Interface

Haskell has a *foreign function interface* (FFI) that allows Haskell code to call, and be called by, foreign language code (primarily C) [9]. Foreign languages also have their own threading models — in C there is POSIX or Win32 threads, for example — so we need to specify how Concurrent Haskell interacts with the threading models of foreign code.

The details of the design can be found in Marlow et al. [6], in the following sections we summarise the behaviour the Haskell programmer can expect.

All of the following assumes that GHC's `-threaded` option is in use. Without `-threaded`, the Haskell process uses a single OS thread only, and multi-threaded foreign calls are not supported.

15.1 Threads and foreign out-calls

An out-call is a call made from Haskell to a foreign language. At the present time the FFI supports only calls to C, so that's all we describe here. In the following we refer to threads in C (i.e. POSIX or Win32 threads) as "OS threads" to distinguish them from Haskell threads.

As an example, consider making the POSIX C function `read()` callable from Haskell:

```
foreign import ccall "read"
   c_read :: CInt      -- file descriptor
          -> Ptr Word8 -- buffer for data
          -> CSize     -- size of buffer
          -> CSSize    -- bytes read, or -1 on error
```

This declares a Haskell function `c_read` that can be used to call the C function `read()`. Full details on the syntax of `foreign` declarations and the relationship between C and Haskell types can be found in the Haskell report [9].

Just as Haskell threads run concurrently with each other, when a Haskell thread makes a foreign call, that foreign call runs concurrently with the other Haskell threads, and indeed with any other active foreign calls. Clearly the only way that two C calls can be running concurrently is if they are running in two separate OS threads, so that is exactly what happens: if several Haskell threads call `c_read` and they all block waiting for data to be read, there will be one OS thread per call blocked in `read()`.

This has to work despite the fact that Haskell threads are not normally mapped one-to-one with OS threads; as we mentioned earlier (Section 8), in GHC, Haskell threads are lightweight and managed in user-space by the runtime system. So to handle concurrent foreign calls, the runtime system has to create more OS threads, and in fact it does this on demand. When a Haskell thread makes a foreign call, another OS thread is created (if necessary), and the responsibility for running the remaining Haskell threads is handed over to the new OS thread, meanwhile the current OS thread makes the foreign call.

The implication of this design is that a foreign call may be executed in *any* OS thread, and subsequent calls may even be executed in different OS threads. In most cases this isn't important, but sometimes it is: some foreign code must be called by a *particular* OS thread. There are two instances of this requirement:

- Libraries that only allow one OS thread to use their API. GUI libraries often fall into this category: not only must the library be called by only one OS thread, it must often be one *particular* thread (e.g. the main thread). The Win32 GUI APIs are an example of this.
- APIs that use internal thread-local state. The best-known example of this is OpenGL, which supports multi-threaded use, but stores state between API calls in thread-local storage. Hence, subsequent calls must be made in the same OS thread, otherwise the later call will see the wrong state.

For this reason, the concept of *bound threads* was introduced. A bound thread is a Haskell thread/OS thread pair, such that foreign calls made by the Haskell thread always take place in the associated OS thread. A bound thread is created by `forkOS`:

```
forkOS :: IO () -> IO ThreadId
```

Care should be taken when calling `forkOS`: it creates a complete new OS thread, so it can be quite expensive.

15.2 Threads and foreign in-calls

In-calls are calls to Haskell functions that have been exposed to foreign code using `foreign export`. For example, if we have a function `f` of type `Int -> IO Int`, we could expose it like this:

```
foreign export ccall "f" f :: Int -> IO Int
```

This would create a C function with the following signature:

```
HsInt f(HsInt);
```

here `HsInt` is the C type corresponding to Haskell's `Int` type.

In a multi-threaded program, it is entirely possible that `f` might be called by multiple OS threads concurrently. The GHC runtime system supports this (at least with `-threaded`), with the following behaviour: each call becomes a new *bound thread*. That is, a new Haskell thread is created for each call, and the Haskell thread is bound to the OS thread that made the call. Hence, any further out-calls made by the Haskell thread will take place in the same OS thread that made the original in-call. This turns out to be important for dealing with GUI callbacks: the GUI wants to run in the main OS thread only, so when it makes a callback into Haskell, we need to ensure that GUI calls made by the callback happen in the same OS thread that invoked the callback.

15.3 Further reading

- The full specification of the Foreign Function Interface (FFI) can be found in the Haskell 2010 report [9];

- GHC's extensions to the FFI can be found in the GHC User's Guide[36];
- Functions for dealing with bound threads can be found in the documentation for the `Control.Concurrent` module.

16 Using concurrency to exploit parallelism

ToDo: perhaps an example that involves non-determinism and therefore can't use the deterministic parallel programming models.

17 Distributed concurrency

Up until now, we have been considering programs that run on a single machine, while possibly making use of multiple processors to exploit parallelism. There is a far more plentiful source of parallelism available though: running a program on multiple *machines* simultaneously. We call this *distributed* programming, and Haskell provides a framework called `remote`[37] that supports it.

Aside from the obvious advantages of multi-machine parallelism, there are other reasons to want to write distributed programs. For example:

- A distributed server can make more efficient use of network resources by moving the servers closer to the clients. We shall see an example of this in Section 17.11.
- A distributed program can exploit a heterogenous environment, where certain resources are only available to certain machines. An example of this might be a cluster of machines with local disks, where a large data structure is spread across the disks and we wish to run our computation on the machine that is has the appropriate part of the data structure on its local disk.

So what should distributed programming look like from the programmer's perspective? Should it look like Concurrent Haskell, with `forkIO`, `MVar` and `STM`? In fact there are some good reasons to treat distributed computation very differently from computation on a shared-memory multicore:

- There is a realistic possibility of partial hardware failure: that is, some of the machines involved in a computation go down while others continue to run. Indeed, given a large enough cluster of machines, having nodes go down becomes the norm. It would be unacceptable to simply abort the entire program in this case. Recovery is likely to

[36] `http://www.haskell.org/ghc/docs/latest/html/users_guide/`
[37] Also known as "Cloud Haskell"

be application-specific, so it makes sense to make failure visible to the programmer and let them handle it in an appropriate way for their application.

- Communication time becomes non-trivial. In the shared-memory setting it is convenient and practical to allow unrestricted sharing, since for example passing a pointer to a large data structure from one thread to another really does have no cost (beyond the costs imposed by the hardware and the runtime memory manager, but again it is convenient and practical to ignore these). In a distributed setting however, communication can be costly, and sharing a data structures between threads is something that the programmer will want to have explicit control over.

- In a distributed setting it becomes impractical to provide any global consistency guarantees of the kind that, for example, STM provides in the shared-memory setting.

For these reasons the conclusion we came to is that the model for distributed programming should be based on explicit *message passing*[38], and not the `MVar` and `STM` models that we provide for shared-memory concurrency. Think of it as having `TChan` be the basic primitive available for communication. It is possible to build higher-level abstractions on top of the explicit message-passing layer, and we shall consider how to do this later.

17.1 The `remote` framework

There is no special runtime support for distribution in Haskell: it is all implemented as a library using the existing concurrency primitives and the `network` library. The examples in this chapter are based around the `remote-0.1.1` package which is available on Hackage, and you will also need the `derive` package for automatically deriving `Binary` instances.

At the time of writing, the `remote` framework is somewhat new and a little rough around the edges, but it is already quite fully-featured and we expect it to mature in due course.

It is reasonable to wonder whether we even need a framework to do distributed message-passing. After all, can't we just use the `network` package directly and program our own message-passing? Certainly you could do this, but the `remote` package provides a lot of functionality that makes it much easier to build a distributed application. Think of it this way: we want to write a *single program that happens to run on multiple machines*, rather than a collection of programs running on different machines that talk to each other. For example, with the `remote` framework we can call a function `spawn` that spawns a process (like a thread) on a different machine, and

[38] also known as the *actor model*

exchange messages with the remote process directly in the form of Haskell datatypes. Even though we are writing a single program to execute on multiple machines, there is no need for all the machines to be identical: indeed it is often the case that we want to exploit some non-uniformity, for example to perform a database query on a machine that is closer to the database itself.

The `remote` frameork provides a whole suite of infrastructure that supports the distributed application domain. Some of the important facilities it provides are:

- remote spawning of processes,
- serialisation of Haskell data for message-passing,
- process linking (receiving notification when another process dies),
- receiving messages on multiple channels,
- a distinguished per-process channel for receiving dynamically-typed messages,
- automatic peer discovery.

Furthermore, the `remote` API is designed to be independent of the actual transport layer being used to communicate between nodes. In fact only TCP/IP is implemented currently, but in the future we expect the `remote` layer to have separate back-ends for Infiniband and other high-speed transports, while the external API would remain identical (modulo some way to select which back-end to use).

17.2 Distributed concurrency or parallelism?

We have included distribution in the concurrency part of this book, for the simple reason that the explicit message-passing API we shall be describing is concurrent and nondeterministic. And yet, the main reason to want to use distribution is to exploit the *parallelism* in running on multiple machines simultaneously, so it is a little unfortunate that we have to resort to a nondeterministic programming model to achieve that. We consider how it might be possible to build deterministic parallelism on top of the distributed message-passing API in Section 17.12.

17.3 A first example: pings

To get acquainted with the basics of distributed programming, we will start with a simple example: a ping/pong message exchange. To start with there will be a single master process, and it will create a child process. The master

```
data ProcessM  -- instance Monad, MonadIO

data NodeId    -- instance Eq, Ord, Typeable, Binary
data ProcessId -- instance Eq, Ord, Typeable, Binary

getSelfPid  :: ProcessM ProcessId
getSelfNode :: ProcessM NodeId

spawn  :: NodeId -> Closure (ProcessM ()) -> ProcessM ProcessId

send   :: Serializable a => ProcessId -> a -> ProcessM ()
expect :: Serializable a => ProcessM a

terminate :: ProcessM a

say :: String -> ProcessM ()

remoteInit :: Maybe FilePath
           -> [RemoteCallMetaData]
           -> (String -> ProcessM ())
           -> IO ()
```

Figure 12: Basic `Remote` API

process will send a "ping" message to the child, which will respond with a "pong" message, and the program will then exit.

The ping example will illustrate the basic pattern for setting up a program to use the `remote` framework, and introduce the APIs for creating processes and simple message passing. The first version of the program will run on a single *node* (machine), so that we can get familiar with the basics of the interface before moving on to working with multiple nodes.

For reference, the subset of the `Remote` API that we will be using is shown in Figure 17.3.

17.3.1 Processes and the `ProcessM` monad

First a bit of terminology: a distributed program consists of a set of *processes* that may communicate with each other by sending and receiving messages. A process is like a thread: processes run concurrently with each other, just like threads, and every process has a `ProcessId` which is distinct from other processes. There are a couple of important differences between threads and processes, however:

- A process can be created on a remote node, whereas threads are always created on the local node (we won't be using this facility until the next section, though).
- Processes run in the `ProcessM` monad, rather than the `IO` monad.

`ProcessM` is an instance of `MonadIO`, so you can perform `IO` operations in `ProcessM` by wrapping them in `liftIO`. All of the message-passing operations are in `ProcessM`, so only processes, not threads, can engage in message-passing.

17.3.2 Defining a message type

We start by defining the type of messages that our processes will send and receive[39]:

```
data Message = Ping ProcessId
             | Pong ProcessId
  deriving Typeable

$( derive makeBinary ''Message )
```

The `Ping` message contains the `ProcessId` of the process that sent it, so that the target of the message knows where to send the response (`ProcessId` is the obvious analogue of `ThreadId` for processes). The `Pong` response also includes the `ProcessId` of the responder, so that the master process can tell which process a particular response comes from.

Messages must be instances of two type classes: `Typeable` and `Binary`. This is just a requirement of the framework, which uses the `Binary` class to implement the serialisation of Haskell data types into bits and bytes, and deserialisation back into Haskell data again. The `Typeable` constraint is required because messages can be sent to dyanmically-typed channels (more about this later).

For `Typeable` we can derive the instance directly (line 3), but there is no built-in support for deriving `Binary`. Fortunately the `derive` package provides automatic deriving of `Binary` instances using Template Haskell, and on line 5 we see a Template Haskell splice to achieve exactly that (don't forget to `import Data.DeriveTH` first, though).

`Typeable` and `Binary` are normally packaged up together and referred to as `Serializable`, using the following class provided by `Remote`:

```
class (Binary a, Typeable a) => Serializable a
instance (Binary a, Typeable a) => Serializable a
```

There's nothing magic about `Serializable`; just think of `Serializable a` as shorthand for `(Binary a, Typeable a)`. You'll see `Serializable` used a lot in the `Remote` APIs.

17.3.3 The ping server process

Next, we'll write the code for a "ping server" process. The ping server must (a) wait for a `Ping` message, and (b) respond with a `Pong` message.

[39] the full code is in `remote-ping/ping.hs`

```
pingServer :: ProcessM ()
pingServer = do
  Ping from <- expect
  mypid <- getSelfPid
  send from (Pong mypid)
```

First of all, notice that we are in the `ProcessM` monad. As we mentioned earlier, virtually all of the `Remote` API is in this monad, and only code running in the `ProcessM` monad can communicate with other processes and spawn new processes. There has to be a way to get into `ProcessM` in the first place; we'll see how that happens shortly, but for now let's assume we're already in `ProcessM` and we need to program the ping server.

On line 3, we receive the next message using `expect`:

```
expect :: Serializable a => ProcessM a
```

The `expect` function receives a message sent directly to this process. Each process has a channel associated with it, and the channel can receive messages of any type. The `expect` call receives only a particular type of messages though, and the type of messages it receives is determined by the context. This is rather like Haskell's `read` function that parses a string into a particular type: if the type cannot be determined from the context, you may need to give an explicit type signature. In this case, the type of messages to receive is determined by the pattern match on the result, which here matches directly on the `Ping` constructor and thus forces `expect` to receive messages of the type `Message` that we defined above.

What happens if the channel contains messages of other types? They are simply ignored and left in the channel for the time being. The `expect` function does not necessarily return the message at the front of the queue, because it might have the wrong type. In fact, `expect` might have to search through the queue of messages for the current process to find one of the right type. If there are no messages of the right type, `expect` will block until one arrives. Therefore it should be used with care: the other messages in the queue are ignored while `expect` is waiting for the right kind of message to arrive, which could lead to a deadlock. We'll see later how to wait for several different types of message at the same time.

Now, we are going to invoke `pingServer` in a separate process. Although in this example we will be creating the new process on the local node, in general we might be creating the new process on another node. Functions that will be executed remotely in this way need to be declared explicitly.[40] The following declaration invokes a bit of Template Haskell magic that creates the necessary infrastructure to allow `pingServer` to be executed remotely:

```
$( remotable ['pingServer] )
```

[40] We expect that in the future GHC will provide syntactic sugar to make remote code execution easier.

17.3.4 The master process

Next, we will write the code for the master process, which you can see below. As you might expect, this is an operation of type `ProcessM ()`:

```
1  master :: ProcessM ()
2  master = do
3    node <- getSelfNode
4
5    say $ printf "spawning on %s" (show node)
6    pid <- spawn node pingServer__closure
7
8    mypid <- getSelfPid
9    say $ printf "pinging %s" (show pid)
10   send pid (Ping mypid)
11
12   Pong _ <- expect
13   say "pong."
14   terminate
```

- Line 3 calls `getSelfNode`, which returns the `NodeId` of the current node. A `NodeId` is needed when creating a new process.
- Line 5 prints a debugging message using `say`; this is a useful way to debug your program.
- Lint 6 calls `spawn` to create the child process:

  ```
  spawn :: NodeId -> Closure (ProcessM ()) -> ProcessM
     ProcessId
  ```

  `spawn` creates a new process on the given `NodeId` (which here is the current node). The new process runs the computation supplied as the second argument to `spawn`, which is a value of type `Closure (ProcessM ())`. Ultimately we want to spawn a computation of type `ProcessM ()`, but such values cannot be *serialised*, that is turned into raw data and squirted over the network, because in practice a value of type `ProcessM ()` could refer to an arbitrary amount of local data, including things that cannot be sent to other nodes (such as `TVar`). Hence the type `Closure` is used to represent serialisable values of type `ProcessM ()`. How do we get one of these? Remember above where we used the Template Haskell function `remotable` to declare that `pingServer` could be executed remotely? This generated a value called `pingServer__closure` with type `Closure (ProcessM ())`, that we can pass as the second argument to `spawn`.

  The `spawn` operation returns the `ProcessId` of the new process, which we bind to `pid`.
- Line 8 calls `getSelfPid` to return the `ProcessId` of the current process. We need this to send in the `Ping` message.

- Lines 11 sends the `Ping` message to the child processe, using the function `send`:

```
send :: (Serializable a) => ProcessId -> a -> ProcessM ()
```

- Line 13 calls `expect` to receive the `Pong` message from the child process.

- Finally (lines 14-15), we print a diagnostic message and terminate the process by calling `terminate`. In this case simply returning from `master` would terminate the process, but sometimes we need to end the process in a context where it is not practical to arrange the top-level function to return, and in those cases `terminate` is useful. Moreover it is good practice to indicate the end of the process explicitly.

17.3.5 The `main` function

All that remains to complete the program is to define our `main` function, and here it is:

```
main = remoteInit (Just "config") [Main.__remoteCallMetaData] $
         \_ -> master
```

The `main` function calls `remoteInit` to initialise the `remote` framework and start the node. The first argument to `remoteInit` is the name of a configuration file to read, which we have called `"config"`. The file should contain the following:

```
cfgRole MASTER
cfgHostName localhost
cfgKnownHosts localhost
```

Don't worry about what this means yet, it isn't important for this single-node example.

The second argument to `remoteInit` is the metadata used to execute remote calls; in this case we pass `Main.__remoteCallMetaData` which is generated by the Template Haskell call to `remotable` we showed earlier.

The final argument is a function of type `String -> ProcessM ()`. Here we use the function `\_ -> master` which ignores the `String` argument and calls `master` (the `String` argument is used in a multi-node setting, we'll explain it in the next example).

So essentially the `main` function initialises the `remote` framework and starts a single process which invokes `master`.

Running the program. First make sure that the current directory contains the file `"config"` mentioned above, and then run the program. You should see output like this:

```
$ ./ping
2012-05-09 16:08:38.848328 BST 0 pid://localhost:42337/8/
     SAY spawning on nid://localhost:42337/
2012-05-09 16:08:38.849693 BST 0 pid://localhost:42337/8/
     SAY pinging pid://localhost:42337/9/
2012-05-09 16:08:38.850066 BST 0 pid://localhost:42337/8/
     SAY pong.
```

Summing up. In this section we built the simplest distributed program possible: it spawns a single child process and performs a simple ping-pong message exchange. The key things to take away are:

- To create a process, we call `spawn`, passing a `NodeId` and a `Closure (ProcessM ())`. The former we got from `getSelfNode` (there are other ways, which we will encounter shortly), and the latter is generated by a call to the Template Haskell function `remotable`.
- Processes run in the `ProcessM` monad, which is a layer over the `IO` moand.
- Messages can be sent to a process using `send`, and received by calling `expect`. Messages are ordinary Haskell data, the only requirement is that the type of the message is an instance of the `Binary` and `Typeable` classes.

There is a certain amount of boilerplate associated with distributed programming: deriving `Binary` instances, declaring remotable functions with `remotable`, starting up the framework with `remoteInit` and so on. Remember that the `remote` framework is currently implemented as a library entirely in Haskell. There is no support for distributed programming built into the language or GHC itself, and this accounts for some of the boilerplate. Over time some of the details will probably change and distributed programming will likely become a smoother experience.

17.4 Multi-node ping

The previous example showed how to create a process and exchange some simple messages. Now we will extend the program to be truly distributed: rather than spawning a process on the local node, we will run the program on several nodes, create a process on each one, and perform the ping-pong protocol with all nodes simultaneously.

The `Message` type and `pingServer` remain exactly as before, the only changes will be to the `master` and `main` functions. The new `master` function is shown in Figure 17.4. It looks like there is a lot of code here, but it's really quite straightforward:

```
master :: ProcessM ()
master = do
  peers <- getPeers

  let workers = findPeerByRole peers "WORKER"

  ps <- forM workers $ \nid -> do
          say $ printf "spawning on %s" (show nid)
          spawn nid pingServer__closure

  mypid <- getSelfPid

  forM_ ps $ \pid -> do
    say $ printf "pinging %s" (show pid)
    send pid (Ping mypid)

  waitForPongs ps

  say "All pongs successfully received"
  terminate

waitForPongs :: [ProcessId] -> ProcessM ()
waitForPongs [] = return ()
waitForPongs ps = do
  m <- expect
  case m of
    Pong p -> waitForPongs (filter (/= p) ps)
    _  -> say "MASTER received ping" >> terminate
```

- Line 3 calls `getPeers`, which returns a list of `NodeId` representing each of the nodes (other than the master node) involved in the computation. The framework automatically discovers nearby peers, and we'll see shortly how to start up the program on multiple nodes.
- Line 5 filters the list of peers to find those that are designated as `"WORKER"` nodes. Each node has a designated *role*, and here we are using exactly two roles: `"MASTER"` and `"WORKER"`. Our program will have a single node with the role `"MASTER"`, and all the other nodes will be assigned the role `"WORKER"`. We'll see shortly how we assign roles to nodes.
- Lines 7-9 spawn a new process on each of the worker nodes, and binds the resulting list of `ProcessId`s to `ps`.
- Lines 13-15 send the `Ping` message to each of the new processes.
- Line 17 calls `waitForPongs` (defined on lines 22-28) to receive all the pong messages. It is a simple algorithm that removes each `ProcessId` from the list as its pong message is received, and returns when the list is empty.
- When `waitForPongs` returns, we emit a diagnostic and `terminate` as before.

The `main` function is a little different:

```
1 main = remoteInit (Just "config") [Main.__remoteCallMetaData]
      initialProcess

3 initialProcess :: String -> ProcessM ()
4 initialProcess "WORKER" = receiveWait []
5 initialProcess "MASTER" = master
```

Remember that the third argument to `remoteInit` is a function that takes a `String` as an argument? Well, the `String` represents the role of the current node, and the function can pattern-match on the `String` to decide how to behave, basd on the role. In this case, if the role is `"MASTER"` we call the `master` function, and if it is `"WORKER"` then we want to wait for the master node to get things going. Here we're calling `receiveWait []`, which is an idiom for blocking indefinitely.

You might be wondering how the role is chosen for a given node. Recall that there is a configuration file called `config` that is read when the program starts, containing this:

```
cfgRole MASTER
cfgHostName localhost
cfgKnownHosts localhost
```

The first line tells the `remote` framework what role to use for the current node. This means that when we start multiple nodes, each one needs its own configuration file with the appropriate setting for `cfgRole`. Alternatively the role can be specified by the command-line flag `-cfgRole=WORKER`, which overrides the corresponding `config` file setting. This may be a more convenient way to set the role, especially when starting multiple nodes on the same machine.

17.4.1 Running with multiple nodes on one machine

First I'll illustrate starting multiple nodes on the same machine, and then progress on to multiple machines.

Let's start by creating two `WORKER` nodes:

```
$ ./ping-multi -cfgRole=WORKER &
[3] 58837
$ ./ping-multi -cfgRole=WORKER &
[4] 58847
```

I used `&` to create these as background processes in the shell. If you're on Windows, just open a few Command Prompt windows and run the program in each one.

Having started the workers, we now start the `MASTER` node:

```
$ ./ping-multi
2012-05-09 16:39:47.522472 BST 0 pid://localhost:39618/8/
     SAY spawning on nid://localhost:34957/
2012-05-09 16:39:47.524933 BST 0 pid://localhost:39618/8/
     SAY spawning on nid://localhost:50781/
2012-05-09 16:39:47.527318 BST 0 pid://localhost:39618/8/
     SAY pinging pid://localhost:34957/9/
2012-05-09 16:39:47.528245 BST 0 pid://localhost:39618/8/
     SAY pinging pid://localhost:50781/9/
2012-05-09 16:39:47.530207 BST 0 pid://localhost:39618/8/
     SAY All pongs successfully received
```

The first thing to note is that the master node automatically found the two worker nodes. The `remote` framework includes some *peer discovery* mechanisms that are designed to automatically locate and connect to other instances running on the same machine or other machines on the local network. We will see later how to control this to avoid accidentally connecting with nodes belonging to other programs.

```
data SendPort a      -- instance of Typeable, Binary
data ReceivePort a

newChannel        :: Serializable a
                  => ProcessM (SendPort a, ReceivePort a)

sendChannel       :: Serializable a
                  => SendPort a -> a -> ProcessM ()

receiveChannel    :: Serializable a
                  => ReceivePort a -> ProcessM a

mergePortsBiased :: Serializable a
                  => [ReceivePort a] -> ProcessM (ReceivePort a)
```

Figure 13: Typed Channels

17.4.2 Running on multiple machines

17.5 Typed Channels

ToDo:

For example:

Note that we don't need a `Pong` message any more. Instead the `Ping` message will contain a `SendPort` on which to send the reply, and the reply is just the `ProcessId` of the sender. In fact in this example we don't really need to send any content back at all—just sending `()` would be enough—but for the purposes of illustration we will send back the `ProcessId`.

```
pingServer :: ProcessM ()
pingServer = do
  Ping chan <- expect
  mypid <- getSelfPid
  sendChannel chan (Pong mypid)
```

```
master :: ProcessM ()
master = do
  peers <- getPeers

  let workers = findPeerByRole peers "WORKER"

  ps <- forM workers $ \nid -> do
          say $ printf "spawning on %s" (show nid)
          spawn nid pingServer__closure

  mypid <- getSelfPid

  ports <- forM ps $ \pid -> do
    say $ printf "pinging %s" (show pid)
    (sendport,recvport) <- newChannel
```

```
    send pid (Ping sendport)
    return recvport

  forM_ ports $ \port -> do
     p <- receiveChannel port
     return ()

  say "All pongs successfully received"
  terminate
```

Note that we can just wait for a response on each channel one after another, and when we have received a response on each channel we know that each of the `pingServer` instances has responded once. This code is simpler than the previous example with the `waitForPongs` function.

On the other hand, if we wanted to use a typed channel to send the `Ping` messages, things get more complicated. We want to do something like this (considering just a single worker for simplicity):

```
do
  (s1,r1) <- newChannel
  spawn nid (pingServer__closure r1)

  (s2,r2) <- newChannel
  sendChannel s1 (Ping s2)

  receiveChannel r2
```

This seems quite natural: we create a channel on which to send the `Ping` message, and give the receive end of the channel to the `pingServer` process when we spawn it.[41] But there's a big problem here: `ReceivePorts` are not `Serializable`, which prevents us passing a `ReceivePort` to the spawned process. GHC will reject the program with a type error.

Why are `ReceivePorts` not `Serializable`? If you think about it a bit, this makes a lot of sense. If a process were allowed to send a `ReceivePort` somewhere else, then the implementation would have to deal with two things: routing messages to the correct destination when a `ReceivePort` has been forwarded (possibly multiple times), and routing messages to *multiple* destinations, because sending a `ReceivePort` would create a new copy. This would introduce a vast amount of complexity in the implementation, and it is not at all clear that it is a good feature to allow. So the `remote` framework explicitly disallows it, which fortunately can be done using Haskell's type system.

This does mean that we have to jump through an extra hoop to fix the code above though. Instead of passing the `ReceivePort` to the spawned process, the spawned process must create the channel and send us back the `SendPort`. Which means we need *another* channel so that the spawned

[41] we haven't discussed passing arguments to closures yet, but in fact there is no problem with arguments: it is part of the magic that `remotable` handles for us.

process can send us back its `SendPort`.

```
  do
    (s,r) <- newChannel  -- throw-away channel
    spawn nid (pingServer__closure s)
    ping <- receiveChannel r

    (sendpong,recvpong) <- newChannel
    sendChannel ping (Ping sendpong)

    receiveChannel recvpong
```

Since this extra handshake is a bit of a hassle, you might well prefer to send messages directly to the spawned process using `send` rather than using typed channels, which is exactly what the example code at the beginning of this section did.

17.5.1 Merging channels

In the example above we waited for a response from each child process in turn, whereas the old `waitForPongs` version processed the messages in the order they arrive. In this case it isn't a problem, but suppose some of these messages required a response. Then we might have introduced some extra latency: if a process towards the end of the list `ps` replies early, then it won't get a response until the master process has dealt with the messages from the other processes earlier in the list, some of which might take a while to reply.

ToDo: complete, showing how to merge channels

17.6 Handling failure

One of the important features of the `remote` framework is that it provides facilities for handling failure, and recovering from it.

Here is a basic example showing how the failure of one process can be caught and acted upon by another process. In our ping example, recall that the `Message` type has two constructors:

```
data Message = Ping ProcessId
             | Pong ProcessId
```

and the code for `pingServer` matches explicitly on the `Ping` constructor:

```
pingServer :: ProcessM ()
pingServer = do
  Ping from <- expect
  mypid <- getSelfPid
  send from (Pong mypid)
```

What will happen if the message is a `Pong`, rather than a `Ping`? Both messages have the type `Message`, so `expect` cannot distinguish: if the context requires a message of type `Message`, then either a `Ping` or a `Pong` will do.

```
receiveWait :: [MatchM q ()] -> ProcessM q

match   :: Serializable a
        => (a -> ProcessM q) -> MatchM q ()

matchIf :: Serializable a
        => (a -> Bool) -> (a -> ProcessM q) -> MatchM q ()
```

Figure 14: Receiving multiple types of message

Clearly in the case of a `Pong` the pattern match will fail, and as usual in Haskell this causes an exception to be thrown. Since there are no exception handlers, the exception will result in the termination of the `pingServer` process.

We can catch this failure using `withMonitor`:

```
withMonitor :: ProcessId -> ProcessM a -> ProcessM a
```

`withMonitor` takes a `ProcessId` to monitor and an action to perform. During the action, any failure of the specified process will result in a special message of type `ProcessMonitorException` being sent to the current process.

To wait for either the `ProcessMonitorException` message or a `Pong`, we need to know how to wait for different types of message at the same time. The basic pattern for this is as follows:

```
receiveWait
  [ match $ \p -> do ...
  , match $ \q -> do ...
  ]
```

where `p` and `q` are patterns that match different types of message. The types of these functions are show in Figure 17.6. The function `receiveWait` waits until any of the `match` functions applies to a message in the queue, and then executes the associated action.

Here is how we monitor the `pingServer` process, and then wait for either a `Pong` message or a `ProcessMonitorException`:[42]

```
withMonitor pid $ do
  send pid (Pong mypid)
  receiveWait
    [ match $ \(Pong _) -> do
       say "pong."
       terminate
    , match $ \(ProcessMonitorException pid reason) -> do
       say (printf "process %s died: %s" (show pid) (show
           reason))
       terminate
    ]
```

[42] the full code is in `remote-ping/ping-fail.hs`

Note that we deliberately send the child a `Pong` message to cause it to fail. Running the program results in this:

```
2012-05-10 15:38:14.063126 BST 0 pid://localhost:33924/8/
     SAY spawning on nid://localhost:33924/
2012-05-10 15:38:14.064075 BST 0 pid://localhost:33924/8/
     SAY pinging pid://localhost:33924/9/
2012-05-10 15:38:14.064661 BST 2 pid://localhost:33924/9/
     SYS Process got unhandled exception Pattern match failure in do expression at p
2012-05-10 15:38:14.06498 BST  0 pid://localhost:33924/8/
     SAY process pid://localhost:33924/9/ died: SrException "Pattern match failure i
```

there was a `SYS` diagnostic to inform us that the process died unexpectedly with an exception, and then we see the message from the master process indicating that it received the notification of the failed process.

Typically when a process dies unexpectedly we want to arrange to restart it in a known good state. This is part of the "let it crash" philosophy mentioned earlier on; the idea that instead of fine-grained exception handlers to cope with particular kinds of failure, we should handle failures by letting the entire process crash, but monitor the process and recover from the crash by restarting the process.

It is worth asking whether having a single `Message` data type for our messages was a good idea in the first place. Perhaps we should have made separate types, as in

```
newtype Pong = Pong ProcessId
newtype Ping = Ping ProcessId
```

The choice comes down to whether we are using typed channels or not. With typed channels we could use only a single message type, whereas using the per-process dynamically typed channel with `send` and `expect` or `receiveWait` we could use multiple message types. Having one type for each message would avoid the possibility of pattern-match failure when matching on a message, but unless we also have a catch-all case to match unrecognised messages, the other messages could be left in the queue for ever, which could amount to an undetected error or deadlock. So there might well be cases where we *want* to match both messages, because one is definitely an error, and so using a single message type would help ensure that we always match on all the possible messages.

Which choice is more appropriate depends on the particular circumstances in your application.

```
data ProcessMonitorException
 = ProcessMonitorException ProcessId SignalReason

data SignalReason
  = SrNormal           -- the process terminated normally

  | SrException String -- the process terminated with an
                       -- uncaught exception, which is given
                       -- as a string

  | SrNoPing           -- the process is not responding to
                       -- pings

  | SrInvalid          -- the process was not running at the
                       -- time of the attempt to establish
                       -- monitoring

withMonitor    :: ProcessId -> ProcessM a -> ProcessM a

monitorProcess :: ProcessId -> ProcessId -> MonitorAction
               -> ProcessM ()

data MonitorAction
  = MaMonitor   -- receive ProcessMonitorException
                -- on termination, as a message

  | MaLink      -- receive ProcessMonitorException
                -- on termination, as an async exception

  | MaLinkError -- receive ProcessMonitorException
                -- on failure only, as an async exception

linkProcess :: ProcessId -> ProcessM ()
```

Figure 15: Monitoring and linking processes

17.7 Why explicit serialisation?

17.8 Controlling peer discovery

17.9 Mixing processes and threads

We saw earlier that distributed programs consist of a collection of communicating processes, and that processes are different from threads in that they may communciate with other (possibly remote) processes. Still, there is nothing to stop a distributed program from also creating local threads and using all the concurrency interfaces that we have seen so far.

Some languages, such as Erlang, take the view that since explicit message-passing is the right model for programming distributed applications, we should use *only* that model, and then our programs will run unchanged on both a shared-memory or a distributed platform. There is undeniable benefit in that approach, and indeed it is entirely possible to write your concurrent Haskell programs using only processes and not threads. However, there are a number of attractive advantages to be had when running in a shared-memory environment (such as `STM`), so we see it as important to have both models available.

In the next section we will extend the chat-server example from Section 14.1 to use distribution, but leaving most of its existing multithreaded architecture in place. The result will be a program using a mixture of threads and processes in quite a natural way.

17.10 State in a distributed program

17.11 A distributed chat server

Consistency

Adding failsafety

17.12 Distributed deterministic parallelism

17.13 Distributed parallel computation

18 Conclusion

We hope you have found this tutorial useful! To recap, here are the main points and areas we have covered.

Haskell provides several different programming models for multiprogramming, broadly divided into two classes: *parallel* programming models where the goal is to write programs that make use of multiple processors to improve performance, and *concurrency* where the goal is to write programs that interact with multiple independent external agents.

The Parallel programming models in Haskell are *deterministic*, that is, these programming models are defined to give the same results regardless of how many processors are used to run them. There are two main approaches: `Strategies`, which relies on lazy evaluatation to achieve parallelism, and the `Par` monad which uses a more explicit dataflow-graph style for expressing parallel computations.

On the Concurrency side we introduced the basic programming model involving threads and `MVar`s for communication, and then described Haskell's support for *cancellation* in the form of asynchronous exceptions. Finally we showed how Software Transactional Memory allows concurrent abstractions to be built compositionally, and makes it much easier to program with asynchronous exceptions. We also covered the use of concurrency with Haskell's Foreign Function interface, and looked briefly at how to program concurrent server applications in Haskell.

References

[1] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 251–264, 2007.

[2] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, 2005.

[3] S Marlow, SL Peyton Jones, A Moran, and J Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285, Snowbird, Utah, June 2001. ACM Press.

[4] Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 96–106, 2006.

[5] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. under submission; draft available at `http://community.haskell.org/~simonmar/bib/monad-par-2011_abstract.html`.

[6] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the Haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 22–32, 2004.

[7] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming*, August 2009.

[8] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, and Phil Trinder. Seq no more: Better strategies for parallel haskell. In *Haskell '10: Proceedings of the Third ACM SIGPLAN Symposium on Haskell*, 2010. URL `http://community.haskell.org/~simonmar/papers/strategies.pdf`.

[9] Simon Marlow (ed.). The Haskell 2010 report, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[10] Bryan O'Sullivan and Johan Tibell. Scalable I/O event handling for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 103–108, 2010.

[11] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96*, pages 295–308. ACM Press, 1996.

[12] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering theories of software construction*, 2002.

[13] Simon Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, AFP'08, pages 267–305. Springer-Verlag, 2009.

[14] P.W. Trinder, K. Hammond, H-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. 8(1):23–60, January 1998.

[15] Greg Wilson, editor. *Beautiful code*. O'Reilly, 2007.