Déjà Fu: A Concurrency Testing Library for Haskell

https://github.com/barrucadu/dejafu

Michael Walker

Department of Computer Science
University of York

March 2016

Abstract

Out of work in the formal verification and model checking field has grown the topic of *systematic concurrency testing* (SCT), also known as *stateless model checking*. This technique allows the reliable, deterministic, and rigorous testing of concurrent programs, and has enjoyed some success in the imperative and object-oriented settings. We propose that the functional world could also benefit from systematic concurrency testing, as GHC Haskell in particular provides a very rich set of concurrency primitives.

We have developed a library for writing testable concurrent Haskell programs, using a typeclass-abstraction to select based on the context of use the concrete implementation to use: the primitives provided by the run-time system, or emulated versions provided as part of a testing framework.

This report discusses the design and implementation of this library, called Déjà Fu, including case studies and further developments from the initial version presented at the 2015 Haskell Symposium.

CONTENTS

1 Introduction . . . 1
- 1.1 Parallelism vs Concurrency . . . 1
- 1.2 Testing Concurrent Programs . . . 2
- 1.3 Scope . . . 2
- 1.4 Contributions . . . 3
- 1.5 Report Roadmap . . . 3

2 Concurrency Abstraction . . . 4
- 2.1 The `MonadConc` Typeclass . . . 4
- 2.2 Software Transactional Memory . . . 9
- 2.3 Memory Model . . . 10

3 Program Execution . . . 11
- 3.1 Primitive Actions . . . 11
- 3.2 Stepwise Execution . . . 15
- 3.3 Scheduling . . . 19

4 Systematic Concurrency Testing . . . 20
- 4.1 Schedule Bounding . . . 21
- 4.2 Partial-order Reduction . . . 21

5 Correctness . . . 25
- 5.1 Correct Execution . . . 25
- 5.2 Correct Testing . . . 26

6 Case Studies . . . 28
- 6.1 auto-update . . . 28
- 6.2 Search Party . . . 31
- 6.3 The Par Monad . . . 32

7 Practical Usage . . . 35
- 7.1 Alternatives to Existing Libraries . . . 35
- 7.2 Integration with Testing Frameworks . . . 36

8 Conclusions, Related & Future Work . . . 38
- 8.1 Related Work . . . 39
- 8.2 Future Work . . . 40

Appendices **42**

A Primitive Actions . . . 43

1 INTRODUCTION

[Déjà Fu is] A martial art in which the user's limbs move in time as well as space, [. . .] It is best described as "the feeling that you have been kicked in the head this way before".

– Terry Pratchett, Thief of Time

Concurrency is notoriously difficult to get right [Yang et al., 2013], sometimes with dire consequences [Leveson and Turner, 1993]. The problem largely stems from the nondeterminism of scheduling: the same program with the same inputs may produce different results depending on the schedules chosen at execution time. This makes it difficult to use traditional testing techniques with concurrent programs, which rely on the result of executing a test to be deterministic. So-called "Heisenbugs" make it difficult to be confident of the correctness of concurrent programs: no bug has been observed during the testing process, but how do we *know* that there aren't any?

Despite the difficulty, concurrency is important for producing many real-world applications. For example, applications with a lot of input and output can be more responsive by executing I/O asynchronously. Concurrency is a useful program structuring technique, and it is here to stay.

There are now a few well-known techniques to avoid concurrency bugs, such as protecting mutable state with locks, and acquiring locks in a consistent order. Exercises like the Dining Philosophers [Dijkstra, 1971] and the Santa Claus Problem [Trono, 1994] allow programmers to explore these topics in small well-understood settings. However, as systems grow, it becomes difficult to think about how different components interact, and it is easy to slip up and introduce a bug.

1.1 *Parallelism vs Concurrency*

It is worth clarifying at this early stage some terminology which will be frequently used throughout this report:

Concurrency
A programming methodology, using concepts such as threads, locks, and mutable variables to structure programs.

Parallelism
An aspect of an implementation, where a multiplicity of hardware components are used to execute distinct pieces of code simultaneously.

Concurrency does not require parallelism, as demonstrated by the single-core, single-processor computers of yore. Similarly, parallelism does not require concurrency, as demonstrated by the data-parallel x86 assembly instructions such as `PMULHUW`, which computes an element-wise multiplication of two vectors, each multiplication in parallel.

Unrestricted concurrency is explicit and *semantically visible* [Peyton Jones et al., 1996]. The interleaved execution of threads, when combined with mutable state, gives rise to nondeterminism. Semaphores and locks can give rise to termination errors in the form of deadlock and livelock. Parallelism, in particular the parallel evaluation of expressions, is *semantically invisible* in a language without side-effects.

Concurrency is often implemented using parallelism. Indeed a concurrency abstraction can be used to guarantee parallelism (given suitable hardware), for example by having the ability to restrict the execution of individual threads to given processor cores.

Parallelism is largely outside the scope of this report, although it does make an appearance in the discussion of relaxed memory in Section 2.

1.2 *Testing Concurrent Programs*

Systematic concurrency testing (SCT) [Flanagan and Godefroid, 2005; Musuvathi and Qadeer, 2007; Musuvathi et al., 2008; Thomson et al., 2014] is a way of tackling the problem of nondeterminism when writing tests. It aims to test a large number of schedules, whilst typically also making use of local knowledge of the program to reduce the number of schedules needed to be confident of an accurate result. By testing many schedules, we can increase our confidence that any bugs which have not been found are unlikely to be exhibited.

SCT overcomes the scheduling problem by forcing a concurrent program to use a scheduler implemented as part of the testing framework: either by overriding the concurrency primitives of the language, or by modifying the program under test to call out to this new scheduler (as in PULSE [Claessen et al., 2009]).

Once the scheduler is under control, schedules can be recorded and replayed, giving reproducibility. Furthermore, by observing which scheduling decisions are available at each decision point, possible schedules can be systematically explored, making different decisions on subsequent executions. Common methods of choosing schedules to take are random [Thomson et al., 2014], schedule bounding [Musuvathi and Qadeer, 2007], and partial-order reduction [Flanagan and Godefroid, 2005]. The latter of these is *complete*: partial-order reduction will find all distinct program states given enough time, in a more intelligent way than just trying all schedules.

1.3 *Scope*

We aim to support all of the functionality of GHC's concurrency API, as made available through the Control.Concurrent and Control.Exception module hierarchies, which (1) does not unavoidably *require* support from the run-time system, and (2) is not so nondeterministic that there is no sensible way to model it accurately.

Some specific examples of things which are out of scope:

- `threadDelay`, as all this guarantees is that a thread will not run *sooner* than the delay. There is no upper bound on the delay, and also no guarantee that any other thread will be scheduled during the delay.

- `threadWaitRead`, `threadWaitWrite`, and the `STM` variants, as there is no way to tell if a file descriptor can be read from or written to without involving `IO`, and in addition is influenced by other non-Haskell processes accessing the same file.
- Threads bound to specific operating-system threads ("bound threads"), as these affect which operating system thread FFI calls operate on, and so alter program behaviour in a parallel setting.
- `BlockedIndefinitely` exceptions, as determining if a thread is blocked indefinitely on an `MVar` is a garbage collection problem, which is out of the reach of the programmer. Déjà Fu does provide some annotation functions to record which shared state a thread knows about, and so similar functionality can be supported on a limited scale, even the GHC documentation warns programmers against relying on these exceptions for correct functioning of a program.[1]

1.4 *Contributions*

Our contributions can be seen as follows:

- Existing results from the concurrency testing world have been applied to functional programming.
- A somewhat novel partial-order reduction algorithm for systematic concurrency testing based on a combination of two others: bounded partial-order reduction Coons et al. [2013] and relaxed-memory DPOR Zhang et al. [2015].
- The Déjà Fu library for testing concurrent Haskell programs, using this algorithm.

1.5 *Report Roadmap*

Firstly we explore the problem of testing concurrent programs and how it can be done. In **Section** 2 we discuss our typeclass abstraction for concurrency and how it relates to GHC's standard concurrency API in terms of functionality. **Section** 3 explains how, given a monadic action polymorphic in the monad (as long as it has an instance of to our typeclass) we can execute it with a given scheduler, and **Section** 4 extends this to cover a systematic exploration of the space of all schedules. **Section** 5 discusses the issues of correctness: how do we know if a result reported by Déjà Fu is actually right?

Then, we move on to the real-world impact of this work, with case studies of Déjà Fu applied to two instances of pre-existing code, and one custom library in **Section** 6. **Section** 7 further discusses the usage of Déjà Fu in combination with existing code. To conclude, **Section** 8 discusses related work, and summarises the community reception to the idea and what is still to be done.

1 *"Note that this feature is intended for debugging, and should not be relied on for the correct operation of your program. There is no guarantee that the garbage collector will be accurate enough to detect your deadlock, and no guarantee that the garbage collector will run in a timely enough manner."* GHC Base Libraries [2015]

2 CONCURRENCY ABSTRACTION

For readers who are unfamiliar with Haskell, syntax and terminology will be explained as they are introduced.

In order to test concurrent programs, we must first create a concurrency abstraction rich enough to express everything that is commonly needed. However, care must be taken that it does not become so rich that it becomes too difficult to implement.

The abstraction developed for Déjà Fu can be thought of as three independent components:

1. There is the `MonadConc` typeclass,[2] which abstracts over much of the operations provided in the Control.Concurrent hierarchy,[3] and also some other functionality like mutable memory cells (`IORef`s) and exceptions.

2. There is the `MonadSTM` typeclass, which abstracts over GHC's software transactional memory API, and is related to `MonadConc` but can be used independently.

3. There is the memory model, which influences the behaviour of some of the `MonadConc` operations and also the SCT behaviour.

Originally only a sequentially consistent memory model was provided, but some support for relaxed memory was added following community feedback.

2.1 *The `MonadConc` Typeclass*

Readers already familiar with GHC's concurrency primitives may find it enough to skim this section noting the syntactic differences in the Déjà Fu variant.

DEPARTURE The few departures from the semantics of the traditional concurrency abstraction are highlighted like this. □

The `MonadConc` typeclass has an instance[4] for `IO`, and so existing code using no I/O other than concurrency and exceptions can be made suitable for testing quite simply. Existing code which makes use of more functionality may require a light dusting of `liftIO`[5] where it is safe, see §2.1.

Threads

Threads let a program do multiple things at once. Every program has at least one thread, which starts where `main` does and runs until the program terminates. A thread is the basic unit of concurrency. It lets us pretend (with parallelism, it might even be true!) that we're computing multiple things at once.

2 Typeclasses are similar to interfaces in object-oriented languages. The key difference is that they also allow polymorphism based on the *return type* of a function as well as the *argument types*.
3 Haskell modules are arranged into a hierarchy, corresponding to files and directories.
4 A typeclass has *instances*; each type may have one unique instance for a typeclass.
5 The `IO` type allows unrestricted side-effects during execution. It turns out that many useful types are just `IO` with some extra structure applied, and the `liftIO` function (which belongs to a typeclass called `MonadIO`) can be used to 'translate' the effects into such a type.

We can start a new thread with the function:[6]

```haskell
fork :: MonadConc m => m () -> m (ThreadId m)
```

The `fork` function starts evaluating its argument in a separate thread. It also gives us back a (monad-specific) `ThreadId` value, which we can use to kill the thread later on, if we want.

A thread can query its own `ThreadId`:

```haskell
myThreadId :: MonadConc m => m (ThreadId m)
```

In a real machine, there are of course a number of processors and cores. It may be that a particular application of concurrency is only a net gain if every thread is operating on a separate core, so that threads are not interrupting each other. The GHC run-time system refers to the number of Haskell threads that can run truly simultaneously as the number of *capabilities*. We can query this value, and fork threads which are bound to a particular capability:

```haskell
getNumCapabilities :: MonadConc m => m Int
forkOn :: MonadConc m => Int -> m () -> m (ThreadId m)
```

The `forkOn` function interprets the capability number modulo the value returned by `getNumCapabilities`.

DEPARTURE `getNumCapabilities` is not required to return a true result. The testing instances return "2" despite executing everything in the same capability, to encourage more concurrency. This is to preserve determinism between different executions, where the actual number of capabilities may differ. The `IO` instance does return a true result. □

Sometimes we just want the special case of evaluating something in a separate thread, for which we can use `spawn` (implemented in terms of `fork`):

```haskell
spawn :: MonadConc m => m a -> m (CVar m a)
```

A `spawn` application returns a `CVar` (*Concurrent Variable*), to which we can apply `readCVar`, blocking until the computation is done and the value is stored.

Threads are scheduled non-deterministically. Every time the run-time system decides to perform a context switch, one of the runnable threads will be executed. Sometimes, however, a thread may be runnable but also waiting for something to happen. The programmer can provide a clue to the scheduler that another thread should be tried instead:

```haskell
yield :: MonadConc m => m ()
```

Calling `yield` gives any other thread the opportunity to execute instead of the yielding one, but it is not *required* to cause a context switch except on co-operative multitasking systems.

6 This is a function named `fork` with a *type signature*. Type signatures may contain typeclass constraints, type variables, type constructors (similar to generics in other languages), and concrete types. Here `ThreadId` is a type constructor and is applied to the type variable `m`, which is constrained to be a type with an instance of `MonadConc`.

Threading and the Foreign Function Interface

In order to accommodate Foreign Function Interface (FFI) calls which may block, GHC provides a mechanism for forking a Haskell thread to an operating system thread. This allows FFI calls to be managed by the operating system, unlike normal Haskell threads which are managed by the run-time system and multiplexed onto a smaller number of operating system threads. This means that blocking FFI calls do not necessarily block the entire program.

There is no `MonadConc` equivalent of bound threads, as there would be no way to reliably test their behaviour. Unfortunately, if bound threads are required, `IO` will have to be used.

A few predicates are provided for compatibility:[7]

```haskell
rtsSupportsBoundThreads :: Bool
rtsSupportsBoundThreads = False

isCurrentThreadBound :: MonadConc m => m Bool
isCurrentThreadBound = return False
```

Mutable State

Threading by itself is not really enough. We need to be able to *communicate* between threads: we've already seen an instance of this with the `spawn` function.

The simplest type of mutable shared state provided is the `CRef` (*Concurrent Reference*). `CRefs` are shared variables which can be written to and read from:

```haskell
newCRef    :: MonadConc m => a -> m (CRef m a)
readCRef   :: MonadConc m => CRef m a -> m a
modifyCRef :: MonadConc m => CRef m a -> (a -> (a, b)) -> m b
writeCRef  :: MonadConc m => CRef m a -> a -> m ()
```

The `modifyCRef` function is atomic. The `readCRef` and `writeCRef` functions are not synchronised: it is possible for one thread to read from a `CRef` strictly after another thread has written to it and to observe an old value! See §2.3. To ensure that every thread sees a value as soon as it is written there is a synchronised write function:

```haskell
atomicWriteCRef :: MonadConc m => CRef m a -> a -> m ()
```

However, synchronisation can slow down execution in a parallel environment. Note that `modifyCRef` is also synchronised.

Compare-and-swap (CAS)

As `CRefs` correspond very closely to mutable memory locations, there is also a compare-and-swap interface available. Compare-and-swap is a synchronised atomic primitive which is used to update a location in memory if and only if it has not been changed since some witness value was produced. This role of this witness value is called a `Ticket` here:

7 Function application in Haskell uses no special syntax to denote argument values, only juxtaposition, so `return False` is applying the value `False` to the function `return`. The function `return` is used to inject a value into a monad, it is unfortunately named and has nothing to do with the return keyword in other languages.

```
readForCAS :: MonadConc m => CRef m a -> m (Ticket m a)
peekTicket :: MonadConc m => Ticket m a -> m a
```

A `Ticket` can be used to check if a `CRef` has been written to since it was produced, and can also be used to get the value that was seen then.

```
casCRef :: MonadConc m => CRef m a -> Ticket m a -> a -> m (Bool, Ticket m a)
```

The `casCRef` function is synchronised, and strict in the value written. It replaces the value within a `CRef` if it hasn't been modified since the `Ticket` was produced. It returns an indication of success and a `Ticket` to use in future operations. This operation is often used in the implementation of lock-free synchronisation primitives.

There is also an equivalent of `modifyCRef` using a compare-and-swap. This behaves almost the same as the non-CAS version but may be more performant in some cases, and is strict in the value being written:

```
modifyCRefCAS :: MonadConc m => CRef m a -> (a -> (a, b)) -> m b
```

Mutual Exclusion

A `CVar` is a shared variable under *mutual exclusion*. It has two possible states: *full* or *empty*. Writing to a full `CVar` blocks until it is empty, and reading or taking from an empty `CVar` blocks until it is full. There are also non-blocking functions which return an indication of success:

```
newEmptyCVar :: MonadConc m => m (CVar m a)
putCVar      :: MonadConc m => CVar m a -> a -> m ()
readCVar     :: MonadConc m => CVar m a -> m a
takeCVar     :: MonadConc m => CVar m a -> m a
tryPutCVar   :: MonadConc m => CVar m a -> a -> m Bool
tryTakeCVar  :: MonadConc m => CVar m a -> m (Maybe a)
```

Unfortunately, the mutual exclusion behaviour of `CVar`s means that computations can become *deadlocked*. For example, deadlock occurs if every thread tries to take from the same `CVar`. The GHC run-time system can detect this in some situations (and will complain if it does), and so can Déjà Fu in a more informative way.

DEPARTURE Déjà Fu can only detect deadlock to the same extent as GHC if every thread is annotated with a declaration of the `CVar`s it knows about. This is because GHC can detect deadlocks during garbage collection, which is out of the reach of Déjà Fu. □

Exceptions

Exceptions are a way to bail out of a computation early. Whether they're a good solution to that problem is a question of style, but they can be used to jump quickly to error handling code when necessary. The basic functions for dealing with exceptions are:

```
catch :: (Exception e, MonadConc m) => m a -> (e -> m a) -> m a
throw :: (Exception e, MonadConc m) => e -> m a
```

Calling `throw` causes the computation to jump back to the nearest enclosing `catch` capable of handling the particular exception. As exceptions belong to a typeclass, rather than being a concrete type, different `catch` functions can be nested, to handle different types of exceptions.

DEPARTURE The `IO` `catch` function can catch exceptions from pure code. This is not true in general for `MonadConc` instances. So some things which work normally may not work in testing, and we risk false negatives. This is a small cost, however, as exceptions from pure code are things like pattern match failures and evaluating `undefined`, which are arguably bugs. □

Exceptions can be used to kill a thread:

```
throwTo :: (Exception e, MonadConc m) => ThreadId m -> e -> m ()
killThread :: MonadConc m => ThreadId m -> m ()
```

These functions block until the target thread is in an appropriate state to receive the exception. A thread has a *masking state*, which can be used to block exceptions from other threads. There are three masking states: *unmasked*, in which a thread can have exceptions thrown to it; *interruptible*, in which a thread can only have exceptions thrown to it if it is blocked; and *uninterruptible*, in which a thread cannot have exceptions thrown to it. When a thread is started, it inherits the masking state of its parent. We can also execute a subcomputation with a new masking state:[8]

```
mask :: MonadConc m => ((forall a. m a -> m a) -> m b) -> m b
uninterruptibleMask :: MonadConc m => ((forall a. m a -> m a) -> m b) -> m b
```

In both cases, the action evaluated is passed a function to reset the masking state to the original one. A thread can be forked and given a function to reset the masking state:

```
forkWithUnmask :: MonadConc m => ((forall a. m a -> m a) -> m ())
  -> m (ThreadId m)
forkOnWithUnmask :: MonadConc m => Int -> ((forall a. m a -> m a) -> m ())
  -> m (ThreadId m)
```

We can also fork a thread and call a supplied function when the thread is about to terminate, which is useful for informing the parent when a child terminates, for example:

```
forkFinally :: MonadConc m => m a -> (Either SomeException a -> m ())
  -> m (ThreadId m)
```

The `SomeException` type is the top of the exception hierarchy, and so can be used to catch all exceptions.

8 These functions have a *higher-ranked* type. Removing the `forall` stuff, we have `(m a -> m a) -> m b`, which is a function which takes a function as an argument and returns a result. The `forall` is necessary because, without it, the concrete type that the variable `a` is unified with is fixed across *all* usage sites, whereas with the `forall` it can be determined uniquely everywhere it is used.

Lifting Actions into `MonadConc`

If the programmer needs to make use of `IO` actions, rather than `MonadConc` actions, then this can be achieved by adding a `MonadIO` context and using `liftIO`. However, this can easily compromise the results of testing, as the test runner cannot peek inside `IO` actions. Thus, it is only safe to life an `IO` action in this way if:

- *The action is atomic and synchronised.*

  Otherwise the test framework will possibly miss schedules which lead to a bug.

- *The action is deterministic*, when executed as part of a computation with a deterministic schedule.

  Otherwise a fundamental assumption behind the testing methodology is false, and no guarantees about completeness can be made.

- *The action cannot block on the action of another thread.*

  Otherwise test execution may deadlock.

In practice these restrictions are not particularly onerous. The only unsynchronised actions are the `CRef` ones, or primitives exposed by GHC. Testing in any language already assumes that any I/O is deterministic. Atomicity and inter-thread blocking tend not to be issues when communicating with the external environment.

2.2 *Software Transactional Memory*

`CVar`s are nice, until we need more than one, and find they need to be kept synchronised. As we can only claim *one* `CVar` atomically, it seems we need to introduce a `CVar` to control access to `CVar`s! But that would be unwieldy and prone to bugs.

Software transactional memory (STM) [Shavit and Touitou, 1995] is the solution. STM uses `CTVar`s, or *Concurrent Transactional Variables*, and is based upon the idea of atomic *transactions*. An STM transaction consists of one or more operations over a collection of `CTVar`s, where a transaction may be aborted part-way through depending on their values. If the transaction fails, *none of its effects take place*, and the thread blocks until the transaction can succeed. This means we need to limit the possible actions in an STM transaction to those which can be safely undone and repeated, so we have another typeclass, `MonadSTM`.

`CTVar`s, like `CRef`s, always contain a value, as shown in the types of the functions:

```
newCTVar   :: MonadSTM s => a -> s (CTVar s a)
readCTVar  :: MonadSTM s => CTVar s a -> s a
writeCTVar :: MonadSTM s => CTVar s a -> a -> s ()
```

If we read a `CTVar` and don't like the value it has, the transaction can be aborted, and the thread will block until at least one of the referenced `CTVar`s has been mutated:

```
retry :: MonadSTM s => s a
check :: MonadSTM s => Bool -> s ()
```

We can also try executing a transaction, and do something else if it fails:

```
orElse :: MonadSTM s => s a -> s a -> s a
```

The nice thing about STM transactions is that they *compose*. We can take small transactions and build bigger transactions from them, and the whole is still executed atomically. This means we can do complex state operations involving multiple shared variables without worrying!

Each `MonadConc` has an associated `MonadSTM`, and can execute transactions of it atomically:[9]

```
atomically :: MonadConc m => STMLike m a -> m a
```

The instance of `MonadConc` for `IO` uses `STM` as its `MonadSTM`.

STM can also use exceptions:

```
throwSTM :: (Exception e, MonadSTM s) => e -> s a
catchSTM :: (Exception e, MonadSTM s) => s a -> (e -> s a) -> s a
```

If an exception propagates uncaught to the top of a transaction, that transaction is aborted and the exception is re-thrown in the thread.

2.3 *Memory Model*

There are three memory models supported in Déjà Fu:

Sequential Consistency
: This model is the most intuitive. A program behaves as a simple interleaving of the actions in different threads. When a `CRef` is written to, that write is immediately visible to all threads.

Total Store Order (TSO)
: Each thread has a write buffer. A thread sees its writes immediately, but other threads will only see writes when they are committed, which may happen later. Writes are committed in the same order that they are created.

Partial Store Order (PSO)
: A relaxation of TSO where each thread has a write buffer for each `CRef`. A thread sees its writes immediately, but other threads will only see writes when they are committed, which may happen later. Writes to different `CRef`s are not necessarily committed in the same order that they are created.

The memory model only makes a difference for unsynchronised operations, such as `readCRef`, `writeCRef`, and `readForCAS`.

The default memory model for testing is TSO, as that most accurately models the behaviour of modern x86 processors. The use of a relaxed memory model can require a much larger number of schedules to be tested when unsynchronised operations are used. PSO required no additional work to support, and so was also included.

9 Here `STMLike` is a *type family*, it is used to relate the `MonadConc` and `MonadSTM` typeclasses.

3 PROGRAM EXECUTION

This chapter describes the test execution of a concurrent program written using the `MonadConc` abstraction, not the execution of programs using GHC's actual concurrency primitives. Of course, for correctness of testing, there should be a correspondence between these two models, see Section 5.

The execution of a concurrent program is considered to be the sequential stepwise execution of *primitive actions*, the most basic things that a computation can do.

3.1 *Primitive Actions*

Computations are composed out of a continuation monad, defined as follows:

```
newtype M n r s a = M { runM :: (a -> Action n r s) -> Action n r s }
```

The `Action` type is the type of primitive actions. There are a number of primitive actions used to construct the testing instances of `MonadConc`. Each thread of execution consists of a sequence of continuations, terminated by the `AStop` primitive, which has no continuation and signals the termination of the thread.

The type variables `n`, `r`, `s` and `a` are the underlying monad (this must be something allowing mutable state, like `IO` or `ST`); the mutable reference type of that monad; the corresponding `MonadSTM`; and the input type. That is, the `M n r s a` type is a wrapper around a function which, given a continuation, produces a new primitive action.

The `Functor` instance allows applying a function to the input value of the continuation:

```
instance Functor (M n r s) where
  fmap f (M m) = M (\c -> m (c . f))
```

The `Applicative` instance allows injecting a pure value into the `M` type, by constructing a continuation which consumes this value. It also allows extracting a function from one computation, a value from another, and applying them.

```
instance Applicative (M n r s) where
  pure x = M (\c -> AReturn (c x))

  (M f) <*> (M v) = M (\c -> f (\g -> v (c . g)))
```

AReturn ***action***
Execute the given action.

Why is `AReturn` necessary? Why not `\c -> c x` instead? It is because the scheduler in the testing implementation cannot work at a finer granularity than individual primitive actions, whereas in the real implementation, things like exceptions can pre-empt a return. In order to model this possibility, then, `pure` must have a corresponding primitive.

Finally, the `Monad` instance allows sequencing.

```
instance Monad (M n r s) where
  return = pure
  (M m) >>= (M k) = M (\c -> m (\x -> k x c))
```

The full listing of primitive actions is available in Appendix A. Some of the more interesting ones will be expanded upon in this section.

Threading

All of the forking functions are implemented with the same primitive action, which corresponds most closely to `forkWithUnmask`:[10]

```
fork      ma = forkWithUnmask     (\_ -> ma)
forkOn c ma = forkOnWithUnmask c (\_ -> ma)
```

The `forkWithUnmask` function uses the `AFork` primitive, which creates a new thread:

```
forkWithUnmask (M ma) = M (AFork "" action) where
  action unmask = runM (ma unmask) (\_ -> AStop)
```

AFork ***name (unmask*** $\rightarrow$ ***action) (thread_id*** $\rightarrow$ ***action)***
: Create a new thread from the first action, and continue executing the current thread with the second.

As the testing implementation does not run things in parallel, the `forkOn` variant simply ignores its argument:

```
forkOnWithUnmask _ = forkWithUnmask
```

The name argument to `AFork` is used, if set, to give more helpful execution traces. If no name is given, a unique numeric identifier is used. `MonadConc` provides a variant of each forking function which takes the name as a parameter. Like the non-named functions, these are all implemented in terms of one:

```
forkWithUnmaskN name (M ma) = M (AFork name action) where
  action unmask = runM (ma unmask) (\_ -> AStop)
```

Exceptions

Exceptions can be used to terminate a computation, either in the current thread or a different one. They can also be caught. The presentation in terms of primitive actions is deceptively simple, see §3.2 for the more complex execution details.

```
throw       e = M (\_ -> AThrow e)
throwTo tid e = M (\c -> AThrowTo tid e (c ()))
```

AThrow ***exception***
: Raises an exception in the current thread, terminating the current execution.

Consider the continuation produced within `throw`. It throws away its argument, there is no further action to perform, hence the only possible thing that the execution can do is to terminate the thread.

10 Type signatures have been omitted here as they were provided in §2.1.

AThrowTo *exception action*
: Raises an exception in the other thread, blocking if the other thread has exceptions masked.

When an exception is raised, the thread it is raised within stops whatever it is currently doing, and backtracks to the closest exception handler capable of dealing with that type of exception. If there is no capable handler, the thread is terminated.

```haskell
catch (M ma) h = M (ACatching (runM . h) ma)
```

ACatching *(exception → handler) action continuation*
: Registers a new exception handler for the duration of the inner action.

Each thread has a stack of exception handlers, where the `ACatching` primitive pushes a new handler, and `APopCatching` pops. `APopCatching` is added automatically when an `ACatching` primitive is evaluated.

APopCatching *action*
: Remove the exception handler from the top of the stack.

There is one primitive action for entering a new masking state:

```haskell
mask (M mb) = M (AMasking MaskedInterruptible (\f -> mb f))

uninterruptibleMask (M mb) = M (AMasking MaskedUninterruptible (\f -> mb f))
```

AMasking *masking_state (unmask → action) continuation*
: Executes the inner action under a new masking state, and also gives it a function to reset the masking state.

Like `ACatching`, the `AMasking` action also has a counterpart primitive to undo its effect, called `AResetMask`. Why is this necessary? `AMasking` can both set and reset the masking state, but the separate function enables more informative execution traces to be generated for the user.

AResetMask *set? inner? masking_state action*
: Sets the masking state.

The `set?` and `inner?` flags are used so that generated traces can helpfully indicate when a `mask` or `uninterruptibleMask` function started and stopped executing, and when an unmasking function passed in to a continuation was used. This is much more helpful for debugging purposes than just seeing that the masking state had been changed.

Software Transactional Memory

STM is implemented with its own set of primitive actions which operate in much the same way as the concurrency primitives. The major difference is that a transaction is executed atomically, whereas the concurrency actions are executed one action at a time, allowing threads to interfere with each other.

```
atomically stm = M (AAtom stm)
```

AAtom ***transaction continuation***
Execute an STM transaction atomically.

When a transaction is executed by `AAtom`, there are three possible outcomes:

1. the transaction completed successfully, and returned a value;
2. the transaction aborted due to an uncaught exception; and
3. the transaction aborted due to a call to `retry`. In this third case, the thread is blocked until any of the `CTVars` referenced in the transaction are mutated, after which it can be tried again.

Transactions can be composed into larger atomic transactions. Aborting a component transaction must not break atomicity. The `orElse` function combines transactions appropriately:

```
orElse a b = S (SOrElse (runSTM a) (runSTM b))
```

SOrElse ***transaction transaction continuation***
Try executing the first transaction, if it fails, execute the second.

As transactions are atomic, the handling of exceptions can be vastly simplified. Catching is performed by executing the entire inner action and inspecting the result. No explicit stack of exception handlers needs to be maintained, as the function call stack suffices.

The effects of a transaction only become visible when it completes successfully. Furthermore, executing a transaction enforces a write barrier.

Testing Annotations

In order for Déjà Fu to perform limited detection of `CVar`- and `CTVar`-based deadlocks, something which GHC can use the garbage collector for, there are the optional testing annotations:

```
_concKnowsAbout var = M (\c -> AKnowsAbout var (c ()))
_concForgets    var = M (\c -> AForgets    var (c ()))
_concAllKnown       = M (\c -> AAllKnown       (c ()))
```

AKnowsAbout ***(Either cvar ctvar) action***
Record that the thread has access to the given variable.

AForgets ***(Either cvar ctvar) action***
Record that the thread no longer has access to the given variable.

AAllKnown ***action***
Record that all variables the thread knows about have been reported.

If a thread is blocked on a `CVar` or `CTVar` (an STM transaction referencing it has aborted), *and* it is known what variables all threads have access to, *and* all other threads with a reference to that variable are also blocked on it, *then* the thread is deadlocked. This can easily be extended to collections of threads which are all blocked on the same variable.

3.2 *Stepwise Execution*

Execution of an entire computation proceeds in a stepwise manner: a thread is chosen by the scheduler, its primitive action is executed, and a new action is returned to be executed by that thread in the next step. The simplest thing that a thread can do is to stop, which will serve as a useful minimal example:

```
stepStop = simple (kill tid threads) Stop
```

The effect of `stepStop` is to: remove the current thread from the map of live threads, and then return the new thread map and the name of the action to appear in the trace (`Stop`, here). The `simple` helper function is for actions which don't create any new shared variables or threads, or have any relaxed-memory effects.

Another simple action that a thread can perform is to return a value:

```
stepReturn c = simple (goto c tid threads) Return
```

The effect of `stepReturn` is to: extract the continuation of the action and replace the continuation of the current thread with it.

Threading

```
type Threads n r s = Map ThreadId (Thread n r s)
```

A map is used to keep track of all current threads. There are helper functions to manipulate this map: `kill` and `goto` are two; another is `launch`, used to create a new thread. This can be seen in the implementation of fork:

```
stepFork name a b = return (Right (threads', idSource', Fork newtid, wb)) where
  threads' = goto (b newtid) tid (launch tid newtid a threads)
  (idSource', newtid) = nextTId name idSource
```

The `stepFork` function involves two modifications to the thread map: firstly, a new thread is created (and inherits the masking state of its parent); secondly the continuation of the current thread is updated. Here `simple` cannot be used, as the identifier source is being modified.

In the implementation of yield, no special functionality is needed:

```
stepYield c = simple (goto c tid threads) Yield
```

Its effect is purely a scheduling concern; from the point of view of updating the state of the system, it is no different to return.

`CRef`s and Relaxed Memory

```
newtype CRef r a = CRef (CRefId, r (Map ThreadId a, Integer, a))
```

A `CRef` is implemented as a mutable reference containing a *globally visible* value, a counter of how many write commits there have been, and a number of *thread-specific* values. These thread-specific values correspond to uncommitted writes. A `Ticket`, used in compare-and-swap operations, is a *witness* that a specific prior value was observed. Like threads, a `CRef` (and `CVar`) can be given a name when it is initially created.

There are three memory models supported by Déjà Fu. Each has a different implementation for writing to a CRef:

```
stepWriteRef cref@(CRef (crid, _)) a c = case memtype of
```

The first model assumes sequential consistency. There are no relaxed memory effects:[11]

```
  SequentialConsistency -> do
    writeImmediate cref a
    simple (goto c tid threads) (WriteRef crid)
```

The writeImmediate function writes to the globally visible value, and clears the thread-specific values.

Total store order (TSO) corresponds to an architecture where each thread has its own cache. This matches modern x86 and x86_64 processors. Writes made by a thread will be cached, but they will be committed in that same order to main memory:

```
  TotalStoreOrder -> do
    wb' <- bufferWrite wb tid cref a tid
    return (Right (goto c tid threads, idSource, WriteRef crid, wb'))
```

The bufferWrite function appends a write to the relevant write buffer, in this case the one corresponding to that thread.

Partial store order (PSO) is a more relaxed version of total store order, where the writes a thread makes may not necessarily be committed in order. It can be modelled by giving each CRef a write buffer, rather than each thread:

```
  PartialStoreOrder -> do
    wb' <- bufferWrite wb crid cref a tid
    return (Right (goto c tid threads, idSource, WriteRef crid, wb'))
```

Both the TSO and PSO cases update the thread-specific map. A thread will always see the writes it has made, but other threads may not.

The compare-and-swap write is a little different. It has the effect of a memory barrier: any uncommitted writes to any CRef are committed before the CAS is done, and the result is immediately globally visible. There is a synchronised function for actions with this barrier property:[12]

```
stepCasRef cref@(CRef (crid, _)) tick a c = synchronised $ do
  (suc, tick') <- casCRef cref tid tick a
  simple (goto (c (suc, tick')) tid threads) (CasRef crid suc)
```

The casCRef function here generates a new Ticket, compares with the supplied one, and then swaps the value. It is provided, rather than the logic be included verbatim, as it is used again in the implementation of stepModRefCas.

The implementation of synchronised is as follows:

11 This is an emple of *do notation*, which is a convenient synctatic sugar for composition of monadic actions.

12 The $ operator is function application, but with a very low precedence. This makes it convenient for avoiding parentheses, which can be more readable when multi-line expressions are involved.

```
synchronised ma = do
  writeBarrier wb
  res <- ma
  case res of
    Right (threads', idSource', act', _) -> return
      (Right (threads', idSource', act', emptyBuffer))
    _ -> return res
```

Here `writeBarrier` commits all cached writes. The action is then executed, and an empty write buffer returned. So `simple` can be used in the implementation of `stepModRef` despite the write buffer being changed.

Cached writes can be committed to the globally visible value (at which point the thread-specific values disappear) by executing a commit action:

```
stepCommit t c = do
  wb' <- case memtype of
    TotalStoreOrder   -> commitWrite wb t
    PartialStoreOrder -> commitWrite wb c
  return (Right (threads, idSource, CommitRef t c, wb'))
```

Note how the invocation of `commitWrite` differs between the cases: under TSO, the cache corresponding to the thread is used; whereas under PSO, the cache corresponding to the `CRef` is used. There is no case for sequential consistency here, as commit actions are not explicitly introduced by the program under test; they are introduced by the test runner when executing under a relaxed memory model. See §3.3.

Reading a reference is quite simple:

```
stepReadRef cref@(CRef (crid, _)) c = do
  val <- readCRef cref tid
  simple (goto (c val) tid threads) (ReadRef crid)

stepReadRefCas cref@(CRef (crid, _)) c = do
  tick <- readForTicket cref tid
  simple (goto (c tick) tid threads) (ReadRefCas crid)
```

The `readCRef` function checks if there is a cached value for the current thread and, if so, returns it. Otherwise it returns the globally visible value. The `readForTicket` function behaves similarly, but returns a `Ticket` rather than the current value.

Exceptions

A thread has both a stack of exception handlers, and a masking state. The handler stack affects all exceptions raised in the thread, whereas the masking state only affects exceptions raised by `throwTo`.

```
stepCatching h ma c = simple threads' Catching where
  a     = runCont ma      (APopCatching . c)
  e exc = runCont (h exc) (APopCatching . c)
  threads' = goto a tid (catching e tid threads)
```

Note the addition of APopCatching at the ends of the enclosed action and the handler. This ensures that the handler is popped from the stack whether an exception is thrown or not.

When an exception is thrown, it may not be able to be handled by the topmost handler, as there are exceptions of many types:

```
stepThrow e = case propagate e tid threads of
    Just threads' -> simple threads' Throw
    Nothing -> return (Left UncaughtException)
```

The propagate function pops from the stack of exception handlers until one is found capable of handling that type of exception. It then jumps to the handler, and returns the new thread map. If no handler was found, the thread is killed by the uncaught exception.

Throwing an exception to another thread is significantly more complicated, and is also a synchronised operation:

```
stepThrowTo t e c = synchronised $
  let threads' = goto c tid threads
      blocked  = block (OnMask t) tid threads
  in if interruptible (lookup t threads)
     then case propagate e t threads' of
            Just threads'' -> simple threads'' (ThrowTo t)
            Nothing
              | t == 0    -> return (Left UncaughtException)
              | otherwise -> simple (kill t threads') (ThrowTo t)
     else simple blocked (BlockedThrowTo t)
```

Firstly, whether the thread is interruptible is checked. If not, the current thread is blocked. If it is interruptible, then the exception is propagated through its handler stack. If a handler is found, the thread jumps to it, throwing away whatever it was going to do next. If a handler is not found, the thread is killed. If the main thread is killed, the entire computation terminates.

Software Transactional Memory

As STM transactions are atomic, the implementation is comparatively simple. They are still implemented in terms of a step function, but it is just iterated until termination.

Firstly, the transaction is executed:

```
stepAtom stm c = synchronised $ do
  (res, idSource') <- runstm stm idSource
  case res of
```

There are now three possible results:

1. The transaction succeeds. All threads blocked on CTVars which were modified are woken:

```
Success _ written val
  let (threads', woken) = wake (OnCTVar written) threads
  in return (Right (goto (c val) tid threads', idSource', STM woken, wb))
```

2. The transaction aborts due to `retry`. The thread is blocked until any of the read `CTVars` are modified:

   ```
   Retry touched ->
     let threads' = block (OnCTVar touched) tid threads
     in return (Right (threads', idSource, BlockedSTM, wb))
   ```

3. The transaction aborts due to an uncaught exception. The exception is thrown in the thread:

   ```
   Exception e -> stepThrow e
   ```

3.3 *Scheduling*

When there are multiple non-blocked threads available, the choice of which one to execute next is made by the scheduler.

A scheduler is represented as a pure function, and is supplied as a parameter when testing. Doing things this way allows for deterministic results and, just as importantly, allows for computing a list of scheduling decisions in advance, designed to try to provoke the system into a new state. This explicit computation of schedules is the basis for the systematic concurrency testing implementation.

```
type Scheduler s = s
  -> Maybe (ThreadId, ThreadAction)
  -> NonEmpty (ThreadId, NonEmpty Lookahead)
  -> (Maybe ThreadId, s)
```

The `Maybe` return value can be used by the schedule to signal that the execution should be aborted. In order to make nontrivial decisions, a scheduler maintains some state, of type `s`. This could be, for example, a random number generator:

```
randomSched :: RandomGen g => Scheduler g
randomSched g _ threads = (Just (threads' !! choice), g') where
  (choice, g') = randomR (0, length threads' - 1) g
  threads'     = map fst (toList threads)
```

The initial state is supplied when the execution begins, and the final state is returned when it terminates. Use of this state is, of course, not mandatory, as a simple round-robin scheduler illustrates:

```
roundRobinSched :: Scheduler ()
roundRobinSched _ Nothing _ = (Just 0, ())
roundRobinSched _ (Just (prior, _)) threads
  | prior >= maximum threads' = (Just (minimum threads'), ())
  | otherwise = (Just (minimum (filter (>prior) threads')), ())
  where threads' = map fst (toList threads)
```

A scheduler is also given information about the state of the system: what the last thread it scheduled did (this is `Nothing` if this is the first step of the computation), and what every runnable thread in the system will do in the next few steps. Here `NonEmpty` is the type of non-empty lists,[13] to give a type-level guarantee that there *are* threads to run: if there are no runnable threads, the execution terminates, signalling a deadlock condition.

The `ThreadAction` type is a record of what has happened. The `Lookahead` type is a slightly simpler view of what will happen. The two types cannot be the same, because in general the effect of performing a primitive action at some point in the future cannot be determined, due to interactions between threads.

Phantom Threads

In a sequentially consistent memory model, the set of runnable threads is exactly the set of threads created by `AFork` which are not blocked.

Under relaxed memory, however, this is not the case. In order to model the nondeterminism of `CRef` writes, for every buffer with an uncommitted write (threads, under TSO; `CRefs`, under PSO), a *phantom thread* is created, and added to the runnable set. A phantom thread is a thread with only one action: `ACommit`. These threads are never added to the thread map, they only exist in order for the scheduler to determine when commits happen.

This may seem like an odd approach: why create new not-quite-threads in order to model relaxed memory? The advantage is that systematic concurrency testing techniques assume there is only one source of nondeterminism: the scheduler. If a second source is added, such as when writes are committed, it is difficult to integrate this with existing algorithms. By using phantom threads, the two sources of nondeterminism are unified, and existing algorithms just work. The phantom thread approach was suggested by [Zhang et al., 2015].

4 SYSTEMATIC CONCURRENCY TESTING

Once the scheduling behaviour of a program can be directed at will, the next step is *systematic* testing of programs for a carefully chosen variety of schedules. Systematic concurrency testing (SCT) comprises a family of techniques, all with the same general aim: to try to find bugs in concurrent programs, more reliably than running a program several times. Within this scope, some techniques are *complete*, in that they find all possible results a program could produce; others are *incomplete*, as there is no such guarantee.

SCT works by providing an initial sequence of scheduling decisions intended to put the program into a new internal state. After this point some deterministic scheduler is used, and the final trace is examined to produce new initial sequences. Typically the assumption is made that all executions are *terminating*: all possible sequences of scheduling decisions will lead to a termination by deadlock or otherwise. Another common assumption is that there is a *finite* number of possible schedules: this forbids finite but arbitrarily long executions, as can be created with constructs such as spinlocks.

Systematic testing terminates when there are no further distinct initial sequences possible.

13 And `toList` converts a `NonEmpty a` to a `[a]`.

4.1 *Schedule Bounding*

Schedule bounding is an *incomplete* approach to SCT. Each sequence of scheduling decisions is associated with a *bound value*, limiting the results of some *bound function*. Such a function could be the number of pre-emptive context switches, for example. Schedule bounding was introduced in [Musuvathi and Qadeer, 2007], and came from work in the model checking field.

Here are three common bound functions in use today:

Pre-emption Bounding [Musuvathi and Qadeer, 2007]
The number of pre-emptive context switches is bounded.

Fair Bounding [Musuvathi and Qadeer, 2008]
The difference between the number of times different threads call `yield` is bounded.

Delay Bounding [Emmi et al., 2011]
The number of deviations from a deterministic scheduler is bounded.

Both pre-emption bounding and delay bounding have empirical evidence, in [Thomson et al., 2014], showing that small bounds are good for finding bugs in many real-world programs.

Fair bounding is used to handle programs which make use of lock-free constructs such as spinlocks. A spinlock may be implemented like so:

```
lock p var = spin where
  spin = do
    x <- readCRef var
    unless (p x) (yield >> spin)
```

Here, a `CRef` is read from repeatedly. Each time some predicate on its value is not satisfied, the thread yields and tries again. This can easily give rise to infinitely long executions: simply don't execute any other thread after the `yield`, as it doesn't *force* a context switch. Fair bounding bounds the difference between the number of times that threads have called `yield`: if the thread that has yielded the fewest times has done so 1 time, and the thread that has yielded the most times has done so 10 times, then the bound value is 9.

Strictly speaking, schedule bounding refers to trying only those schedules with a bound value equal to some fixed parameter. A variant of this is *iterative* bounding, where this parameter is increased from zero up to some limit. Another variant is where an inequality, rather than an equality, is used. This explores the same schedules as iterative bounding, but doesn't impose the same ordering properties over schedules tried. In practice, "schedule bounding" typically refers to this third type, unless specified otherwise.

Déjà Fu uses a combination of pre-emption and fair bounding, with a default pre-emption bound of 2 and a fair bound of 5, in order to handle gracefully any computations which use spinlocking techniques. The default pre-emption bound was chosen based on empirical evidence.

4.2 *Partial-order Reduction*

Partial-order reduction is a *complete* approach to SCT. It is based on the insight that, when comparing different execution traces, only the relative ordering of *dependent* actions is important.

Two actions are dependent if the order in which they are performed could affect the result of the program:

Dependency Relation [Flanagan and Godefroid, 2005]

Let $\mathcal{T}$ be the set of transitions in a concurrent system. A binary, reflexive, and symmetric relation $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{T}$ is a valid *dependency relation* iff, for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin \mathcal{D}$ (t_1 and t_2 are *independent*) following properties hold for all program states s:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'; and
2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.

In other words, independent transitions cannot enable or disable each other, and enabled independent transitions commute. Rather than use this relational definition directly, typically some sufficient conditions for dependency are identified. These conditions are determined by what sorts of things the concurrent system under test can express.

Typically, the presentation of algorithms assumes a very simple core concurrent language of just reads and writes. This gives rise to the following dependency condition:

$$\begin{aligned} x \not\leftrightarrow y \iff & \text{thread\_id}(x) = \text{thread\_id}(y) \vee \\ & (\text{variable}(x) = \text{variable}(y) \wedge (\text{is\_write}(x) \vee \text{is\_write}(y))) \end{aligned}$$

Where $x \not\leftrightarrow y$ is read as "x and y are dependent". This choice of notation would suggest a symbol $\leftrightarrow$ meaning independence, but that doesn't seem to be used.

The dependency relation for Déjà Fu is rather more complex than this, as there are more actions than just reads and writes. However it can be simplified to a few quite general conditions over different sorts of reads and writes, with some remaining special cases.

These special cases are:

```
dependent (t1, a1) (t2, a2) = case (a1, a2) of
  (Lift, Lift)    -> True
  (ThrowTo t, _) -> t == t2
  (_, ThrowTo t) -> t == t1
  (STM _, STM _) -> True
```

- Two lifts from the underlying monad are always dependent, as in general this may allow arbitrary I/O to be performed. The only restriction over I/O is that, given a fixed schedule, the I/O is deterministic.
- Throwing an exception to a thread is dependent with anything, as all actions can be pre-empted by an exception.
- STM transactions are always dependent. This final case could probably be refined to STM transactions which have some overlap in the `CTVars` they use, but this is an optimisation which has not yet been tried.

Furthermore, as a Haskell program terminates when the main thread terminates, there is a dependency between the last action in a trace (whatever it may be) and *everything* else.

The general cases are defined in terms of synchronised and unsynchronised actions. Synchronised actions commit all pending CRef writes.

```
dependentActions memtype buf a1 a2 = case (a1, a2) of
  (UnsynchronisedRead  r1, UnsynchronisedWrite r2) -> r1 == r2
  (UnsynchronisedWrite r1, UnsynchronisedRead  r2) -> r1 == r2
  (UnsynchronisedWrite r1, UnsynchronisedWrite r2) -> r1 == r2
  (UnsynchronisedRead r1, _) | isBarrier a2 -> isBuffered buf r1
  (_, UnsynchronisedRead r2) | isBarrier a1 -> isBuffered buf r2
  _ -> same crefOf && (isSynchronised a1 || isSynchronised a2) || same cvarOf
```

- A read and write to the same CRef are dependent, as are two writes. The reason for this dependence even under a relaxed memory model is because writes give rise to commits, which *do* synchronise.
- An unsynchronised read from a variable is dependent with an action that imposes a memory barrier if there are buffered writes to the same variable.
- Any two actions on the same CRef where at least one of them will cause a commit are dependent.
- Any two actions on the same CVar are dependent.

Characterising the execution of a concurrent program by the ordering of its dependent actions only gives us a *partial order* over the actions in the entire program. An execution trace may be just one possible *total* order corresponding to the same partial order. The goal of partial-order reduction, then, is to eliminate these redundant total orders by intelligently making scheduling decisions to permute the order of dependent actions.

This can be done by executing a program with a deterministic scheduler, and then examining the trace, the total order, for *backtracking points*. A backtracking point is a place in the execution where multiple dependent choices were available, and only one was tried. The exploration of the state space continues by making the same scheduling decisions up to that point, and then making a different decision. This process of doing partial-order reduction based on information gathered at run-time, rather than static analysis, is called *dynamic partial-order reduction* (DPOR).

In an imperative language, DPOR is usually done by executing the program under test stepwise in a recursive function, where each stack frame has a set of decisions still to try, and this is mutated by later calls when a backtracking point is identified. As Déjà Fu is a Haskell library, this is not a very natural way to formulate anything, and so a different approach was taken.

Déjà Fu explicitly constructs a tree in memory, where each path from the root to a leaf corresponds to one complete execution. Forks in the tree correspond to places where multiple decisions have been tried. The operation proceeds like so: generate a list of initial scheduling decisions; execute the computation, using some default scheduler when the supplied decisions are exhausted; add the resultant trace to the tree; search for and add any new backtracking points to the tree; and continue while there is still work to do.

This is implemented as follows:

```
sctBounded memtype bf run = go initialState where
  go state = case next state of
    Just decisions -> do
      (result, s, trace) <- run decisions
      let bpoints = findBacktrack memtype s trace
      let newBPOR = todo bf bpoints (grow memtype trace state)
      ((result, trace) :) <$> go newState
    Nothing -> return []
```

Here `next` returns a schedule prefix; `run` executes the computation with a given sequence of initial scheduling decisions, returning the final result, the final scheduler state (which includes a tentative list of bracktracking points), and the execution trace; `findBacktrack` identifies a list of actual backtracking points from these tentative ones; `grow` adds the trace to the tree structure; and `todo` adds the newly-identified backtracking points. It is also the responsibility of `todo` to ensure these new backtracking points do not cause schedules exceeding the bound to be generated; the `bf` function is the bound function, expressed as a predicate.

The entire process terminates when `next` returns `Nothing`, as there are no unexplored backtracking points left.

Integration with Schedule Bounding

The naïve way to integrate DPOR with schedule bounding would be to first use partial-order techniques to prune the search space, and then to additionally filter things out with schedule bounding.

Unfortunately, this is unsound. This approach misses parts of the search space reachable within the bound. This is because the introduction of the bound introduces new dependencies between actions, which cannot be determined *a priori*. The solution is to add *conservative* backtracking points to account for the bound in addition to any normal backtracking points that are identified. Where to insert these depends on the bound function.

In the case of pre-emption bounding, it suffices to try all possibilities at the last context switch before a normal backtracking point. This is because context switches influence the number of pre-emptions needed to reach a given program state, depending on which thread gets scheduled. As pre-emption bounding has been found empirically to be successful with a low number of threads, and DPOR is already eliminating a lot of possibilities, this is not in practice a huge additional cost.

Integration with Relaxed Memory

Due to the use of phantom threads, explained in §3.3, almost nothing needs to be done to support relaxed memory!

The `dependentActions` function has some knowledge of relaxed memory in order to make less pessimistic decisions, as otherwise the assumption would have to be made that there are always uncommitted writes. The only other change is related to the integration with schedule bounding: a pre-emption immediately before (or immediately after) a phantom thread is free.

5 CORRECTNESS

In order to be useful, a testing tool must be *correct*. More specifically, it needs to be:

Sound
: Every result which the testing functions report as possible can show up under actual execution.

Complete
: The testing functions report as possible every result which can show up under actual execution.

Nothing is missed, and nothing is made up. Comparing test results against multiple actual results can give confidence in the correctness of Déjà Fu, but the whole point of systematic concurrency testing is to move *away* from running things many times! In order to be certain of correctness, a correspondence must be established, formally, between the semantics of the true concurrency abstraction and the testing implementation.

Establishing the correctness of Déjà Fu decomposes very naturally into two parts: is the execution of a computation under a single arbitrary schedule correct; and is the SCT implementation correct?

5.1 *Correct Execution*

Correctness of execution asks whether the result of an arbitrary execution of Déjà Fu's testing implementation can be obtained in reality. Furthermore, do all real-world executions correspond to a possible execution under Déjà Fu? To put it more simply:

- Is the behaviour of the primitive functions the same?
- Is the granularity of scheduling decisions the same?

Both of these come with the caveat that the behaviour can be different, as long as this difference can't be observed.

Primitives

The method of implementing the members of the `MonadConc` typeclass that would be most amenable to proof would be to implement analogues of the GHC primitives directly, and implement everything else in terms of these. This matches how actual Haskell is implemented, and would lend itself to establishing a formal correspondence between the Déjà Fu primitives and the GHC primitives, and the higher-level `MonadConc` functions and the higher-level functions in Control.Concurrent.

This approach was not taken, however. Firstly, it would tie the implementation and correctness of Déjà Fu very closely to the implementation of GHC; in principle the implementation of GHC's concurrency primitives could be completely changed but the behaviour preserved. Secondly, it would restrict Déjà Fu to a very specific type of concurrency, supporting low-level operations, which may not map to all interesting implementations of concurrency.

Instead, a reimplementation of GHC's concurrency based on the documented and observable behaviour of the various functions was done. This allows observing the behaviour of a program and determining, intuitively, whether it is correct or not; but it's not so good for proof. The matter is complicated by there being no standard for concurrent Haskell, there is only what GHC provides.

The correctness of operations using `CRefs` is complicated even further, as the behaviour of these depends on the underlying memory model. Total store order was chosen to be the default, as it is what x86 processors do with unsynchronised memory accesses. But a `CRef` is more complicated than a simple memory cell: it is a pointer to a cell, which can be moved around in garbage collection, and it is accessed through primitive operations more complicated than simple loads and stores. The lack of a standard, or even comprehensive documentation, means that to discover the memory model for `CRefs`, the compilation of `IORef` functions must be traced through GHC from Haskell source to machine code. As GHC uses C-- as an intermediary language, this may also require determining a memory model for C--.

Finally, there are some intended departures from the behaviour of GHC's behaviour documented in §2.1:

- `getNumCapabilities` is not required to return a true result.
- Deadlock detection can only function to the same extent as GHC if every thread is annotated with which `CVars` and `CTVars` it knows about, as Déjà Fu cannot use the garbage collector for this task.
- `catch` is not required to be able to catch exceptions from pure code.

Scheduling

The stepwise implementation allows for a scheduling decision to be made between each primitive action, which doesn't quite correspond to how GHC handles scheduling:

"*GHC implements pre-emptive multitasking: the execution of threads are interleaved in a random fashion. More specifically, a thread may be pre-empted whenever it allocates some memory, which unfortunately means that tight loops which do no allocation tend to lock out other threads (this only seems to happen with pathological benchmark-style code, however).*" GHC Base Libraries [2015]

That is, GHC allows for pre-emptions to occur whilst evaluating pure code, which the stepwise executor does not. So there are executions involving the pre-emption of the evaluation of non-terminating expressions which are possible under GHC but not under Déjà Fu. Whether this can be used to produce different outputs is less clear.

5.2 *Correct Testing*

Correctness of testing asks whether the schedule prefixes generated by the partial-order reduction mechanism are valid, and are there any results possible in real-world execution for which no schedule will be generated? This is definitely *not* the same as asking if every real-world schedule will be generated by the testing framework, as that is precisely what partial-order reduction tries to avoid.

Prefix Validity

Multiple executions with different schedules are stored internally as a tree, with each path from the root to a leaf corresponding to a complete execution:

```
data BPOR = BPOR
  { runnable :: Set ThreadId
  , todo     :: Set Threadid
  , done     :: Map ThreadId BPOR
  , action   :: Maybe ThreadAction
  }
```

The `runnable` field is the set of all threads runnable at that point; `todo` is the decisions still to try making; `done` is the decisions already made; and `action` is what was done at this step. The `action` field has a `Maybe` type, because initially no action has been performed, as the computation hasn't yet started.

There is a unique initial state, where only the initial thread is runnable and nothing has been done:

```
initialState :: BPOR
initialState = BPOR (singleton 0) (singleton 0) empty Nothing
```

There are some basic well-formedness invariants associated with a `BPOR` value:

- Every decision in the to-do set is possible: $\text{todo} \subseteq \text{runnable}$
- Every decision in the done map is possible: $\textbf{dom}\ \text{done} \subseteq \text{runnable}$
- No decision that has been done is in the to-do set: $\text{todo} \cap \textbf{dom}\ \text{done} = \varnothing$
- These properties hold recursively: $\forall p \in \textbf{ran}\ \text{done}.\ \textbf{well\_formed}\ p$

Some work has been started in the Isabelle/HOL theorem prover to formalise part of the recursive loop in `sctBounded` (see §4.2) and to prove that the invariants are preserved, assuming that the stepwise executor is correct. It is hoped that schedule prefix validity will follow from this. Specifically, the things to be proved are:

- A prefix produced by `next` is valid if the `BPOR` tree is well-formed; furthermore, it consists of a sequence of decisions that have already been made and is terminated by a single decision from a to-do set.
- `next` returns `Nothing` if and only if the `todo` field of every node in the `BPOR` tree is empty.
- `grow` adds the information in a trace to the `BPOR` tree, making no other changes.

Data structure invariants are important properties to verify, as if they are broken any assumptions made in the rest of the code cannot be trusted.

Result Completeness

The simplest notion of completeness of interest here is that for all results possible by executing a given program for real, the partial-order reduction framework can give that result.

However, as schedule bounding is involved, this is clearly not the case. Here are two different notions of completeness:

- If the bounds are all set to ∞, all results possible under real execution show up under Déjà Fu execution.
- For all sets of bounds, all results possible under real execution subject to those bounds show up under Déjà Fu execution with the same bounds.

The former corresponds to the correctness of partial-order reduction with no bounds, and does not imply the latter. The latter implies the former, and is the more interesting property. The latter is what we really want of our testing framework.

The proof would need to proceed by first showing that the dependency relation is correct: only actions related by the dependency relation can influence each others results. It's not clear how to approach this, as it relies on the implementation of the actions. Once the correctness of the dependency relation is established, it must be shown that the partial-order reduction only prunes schedules where there is no dependency.

6 CASE STUDIES

Three case studies are discussed, two of which are from pre-existing libraries which have been modified to use the `MonadConc` abstraction. Two known bugs are reproduced in the *auto-update*[14] library. Then, there is a bug which arose unintentionally in the implementation of a library for expressing parallel search problems, where an incorrect use of `CTMVars` allowed a user to observe a partial result. Finally, one of the schedulers in the *monad-par*[15] package is tested.

6.1 *auto-update*

The *auto-update* library runs tasks periodically, but only if needed. For example, a single worker thread may get the time every second and store it to a shared `IORef`, rather than have many threads starting within a second of each other all get the time independently [Snoyman, 2014]. Despite the core functionality being very simple, two race conditions were noticed by users inspecting the code in October 2014.

The entire implementation, excluding comments and imports, is reproduced in Figure 1. The `mkAutoUpdate` function spawns a worker thread, which performs the update action at the given frequency, only if the `needsRunning` flag has been set. It returns an action to attempt to read the current result, if necessary blocking until there is one. The transformation to the `MonadConc` typeclass is mostly simple, however the `threadDelay` must be wrapped inside a call to `liftIO`.

14 https://hackage.haskell.org/package/auto-update
15 https://hackage.haskell.org/package/monad-par

```haskell
data UpdateSettings a = UpdateSettings
    { updateFreq           :: Int
    , updateSpawnThreshold :: Int
    , updateAction         :: IO a
    }

defaultUpdateSettings :: UpdateSettings ()
defaultUpdateSettings = UpdateSettings
    { updateFreq           = 1000000
    , updateSpawnThreshold = 3
    , updateAction         = return ()
    }

mkAutoUpdate :: UpdateSettings a -> IO (IO a)
mkAutoUpdate us = do
    currRef      <- newIORef Nothing
    needsRunning <- newEmptyMVar
    lastValue    <- newEmptyMVar

    void $ forkIO $ forever $ do
        takeMVar needsRunning

        a <- catchSome $ updateAction us

        writeIORef currRef $ Just a
        void $ tryTakeMVar lastValue
        putMVar lastValue a

        threadDelay $ updateFreq us

        writeIORef currRef Nothing
        void $ takeMVar lastValue

    return $ do
        mval <- readIORef currRef
        case mval of
            Just val -> return val
            Nothing -> do
                void $ tryPutMVar needsRunning ()
                readMVar lastValue

catchSome :: IO a -> IO a
catchSome act = catch act $
  \e -> return $ throw (e :: SomeException)
```

Figure 1: *auto-update* implementation

The simpler race condition occurs if the reading thread is pre-empted by the worker thread after putting into `needsRunning`, and does not run again until after the delay has passed. In this case the worker thread can become blocked on taking for a second time from `needsRunning`. The reader thread will be unable to read from `lastValue` as the worker thread emptied it as the last action it performed. The race condition can be exhibited with the following test:

```haskell
test :: (MonadConc m, MonadIO m) => m ()
test = join (mkAutoUpdate defaultUpdateSettings)
```

This test was chosen as it is one of the simplest things a user may wish to do with the library: to create the worker, and to then read the computed value. The output of testing shows the different results that were found, with a sample trace leading to each one:

```
> autocheckIO test
[fail] Never Deadlocks (checked: 1)
        [deadlock] S0--------S1-----------S0-
[pass] No Exceptions (checked: 9)
[fail] Consistent Result (checked: 8)
        [deadlock] S0-----P1-S0---S1-----------S0-
        () S0--------S1---------P0---
False
```

The `autocheckIO` function is used to search for deadlocks, uncaught exceptions in the main thread, and nondeterminism in results. It allows the use of `liftIO`, there is an `autocheck` function which does not.

"Sn" indicates that thread n began executing after the prior one blocked, "Pn" indicates that thread n pre-empted the prior one. It is also possible to obtain a richer data structure with a clearer account of what happened.

To read this trace, it is helpful to look at the "S" points, rather than to count the steps. Using this tactic, we can see that a deadlock occurs if the initial thread runs until it blocks (which is the `readMVar lastValue` call), then thread 1 runs until it blocks (the second time `takeMVar needsRunning` is reached). At this point the initial thread is runnable, as it was unblocked by the write to `lastValue`. This is scheduled, and immediately blocks because thread 1 took the value. Both threads are now blocked, and so the computation is deadlocked.

This deadlock may arise in any use of the library, as it depends only on the timing of the delay, and not on the computation performed. If the call to `threadDelay` completes before the reading thread has resumed execution, this situation will arise.

The more complex race condition arises if `readMVar` isn't atomic, as in GHC versions before 7.8. The `readMVar` function used to be a combination of a take and a put. In this case an old value can be returned if the read of `lastValue` is pre-empted between these two operations, as shown in this test:

```haskell
test :: (MonadConc m, MonadIO m) => m Int
test = do
  var  <- newCRef 0
  let action = modifyCRef var $ \x -> (x+1, x)
  auto <- mkAutoUpdate $ defaultUpdateSettings { updateAction = action }
  auto
  auto
```

Here `auto` is called twice to update the counter variable twice. Actually reproducing this bug requires a new `readCVar` function be written, which has the same behaviour as the old `readMVar` function. Exhibiting this bug requires three pre-emptions. As we need to supply our own bounds, we cannot use `autocheckIO`. The more `dejafuIO'` function allows for the memory model and bounds to be specified:

```
> let bounds = defaultBounds { preEmptionBound = Just 3 }
> dejafuIO' TotalStoreOrder bounds test ("Consistent Result", alwaysSame)
[fail] Consistent Result (checked: 23)
        [deadlock] S0------P1-S0---S1-----------S0-
        0 S0---------S1--------P0-----
        1 S0---------S1---------P0---P1--------P0---
```

Here we see two different (non-deadlocking) results. The issue is the `readMVar` in the main thread and the `tryTakeMVar` followed by `putMVar` in the other. If the main thread takes from the `MVar` before the `tryTake`, it will retrieve the prior value. This in itself is not a problem. However, if the main thread then puts that value back between the `tryTake` and the `put`, the `put` will block, and not be able to store the updated value.

Despite the bugs being rather simple, one not requiring any pre-emptions at all to trigger, they both arose in practice. How easy it is to make mistakes when implementing concurrent programs!

6.2 *Search Party*

The *Search Party*[16] library supports speculative parallelism in generate-and-test search problems. It is motivated by the consideration that if multiple acceptable solutions exist, it may not matter which one is returned. In early versions of the library, only single results could be returned, but support for returning all results was later added, incorrectly, introducing a bug.

The library provides a collection of combinators used to express a generate-and-test problem, which are executed in using a concurrent producer/consumer pattern. One worker thread is created for each capability, which share a list of work items and communicate results back to the main thread.

The key piece of code causing the problem was this part of the worker loop:

```
case maybea of
  Just a -> do
    atomically $ do
      val <- tryTakeCTMVar res
      case val of
        Just (Just as) -> putCTMVar res $ Just (a:as)
        _ -> putCTMVar res $ Just [a]
    unless shortcircuit $
      process remaining res
  Nothing -> process remaining res
```

Here `maybea` is a value indicating whether the computation just evaluated was successful. The intended behaviour is that, if a computation is successful, its result is added to the list in the `res` `CTMVar`. This `CTMVar` is exposed indirectly to the user of the library, as it is blocked upon when the final result of the search is requested.

There are some tests in *Search Party*, checking that deadlocks and exceptions don't arise, and that results which should be deterministic really are. Upon introducing this new functionality,

16 https://github.com/barrucadu/search-party

these tests began to fail with differing result lists returned for different schedules, prompting a new test:[17]

```
checkResultLists :: Eq a => Predicate [a]
checkResultLists = representative (alwaysTrue2 check) where
  check (Right (Just as)) (Right (Just bs)) = as ‘elem‘ permutations bs
  check a b = a == b
```

Given this predicate, we can very clearly see the problem:

```
> dejafu (runFind $ [0..2] @! const True) ("Result Lists", checkResultLists)

[fail] Result Lists (checked: 145)
        Just [0] S0-----S1---------S3-------S0-------
        Just [1] S0-----S1---------S3---P2-------S0-------
        Just [1,0] S0-----S1---------S3-------S2-------S0-------
        Just [0,1] S0-----S1---------S3---P2-------S3----S0-------
False
```

The `@!` operator is a concurrent filter, with the order of results nondeterministic. The problem was a lack of any indication that a list-producing computation had finished. As results were written directly to the `CTMVar`, partial result lists could be read depending on how the worker threads and the main thread were interleaved.

In this case, fixing the failure did not require any interactive debugging. Only one place had been modified in introducing the new functionality, and the bug was found by re-reading the code with the possibility of error in mind. However, the ability to produce a test case which reliably reproduces the problem gives confidence that it will not be accidentally reintroduced.

6.3 *The Par Monad*

The `Par` monad [Marlow et al., 2011] is a library providing a traditional-looking concurrency abstraction, providing the programmer with threads and mutable state, however it maintains determinism by restricting its shared variables to one write, and operations to read block until a value has been written. Thus, `Par`'s `IVar`s are *futures*, not *mutable* state. `Par` uses a work-stealing scheduler running on multiple operating system threads, fully evaluating values on their own threads before inserting them into an `IVar`. Despite its limitations, the `Par` monad can be very effective in speeding up pure code.

The following example maps a function in parallel over a list, fully evaluating it. Of course, laziness is generally what is desired in Haskell programs, but often it is known that an entire result will definitely be needed:

```
parMap :: NFData b => (a -> b) -> [a] -> [b]
parMap f as = runPar $ do
  bs <- mapM (spawnP . f) as
  mapM get bs
```

17 The `representative` function picks only one trace for each unique result.

However, with a lack of multi-write shared variables and non-blocking reads, Par is unsuitable for long-lived concurrent programs with a central shared state. It could not be used to implement a multi-threaded work-stealing scheduler, such as the one underpinning Par itself. The library provides a number of different schedulers, the default being the "trace" scheduler. Due to reports of potential deadlocks with the "direct" scheduler from a year ago, it was tested with Déjà Fu.

To reduce the effort in modifying the code, only the direct dependencies of the "direct" scheduler were modified, the rest of the library being left unchanged. This resulted in four files needing change: two from the *abstract-deque*[18] package and two from the *monad-par*[19] package.

Converting *monad-par* to use Déjà Fu was quite simple. All relevant types were parametrised by the underlying monad, all functions had a MonadConc context added, functions were swapped for their Déjà Fu alternatives, and a runPar' function was added:

```
runPar' :: MonadConc m => Par m a -> m a
```

Some simplifications were made in the conversion process:

- Par normally uses the *mwc-random*[20] package when performing its internal scheduling. This was replaced with a StdGen, as *mwc-random* does not allow the user to provide a seed.
- Behaviour of the Par scheduler can be configured using cpp, but only the default configuration was tested.

Figure 2 shows the original and converted scheduler initialisation code. As can be seen, they are very similar, even though this is a core component of a rather sophisticated library, where the types have been changed.

In total, there were 127 changed lines across the two modules of interest, out of 647 total. 47 of these were type signatures, 10 were type and data declarations, and 4 were typeclass instance declarations. Of the 66 remaining lines, over half were renaming *MVar and *IORef functions to their *CVar and *CRef equivalents. The remaining lines were a combination of import declarations, calls to getNumCapabilities (Déjà Fu does not provide the numCapabilities binding, which was used in several places), and replacing liftIO with lift from the *mtl*[21] package.

The refactoring was driven entirely by resolving type errors, after first parameterising the basic types on the underlying monad, rather than assuming IO. The more general types could even be hidden from the user by providing type synonyms to supply IO as the monad, making the new functionality purely an internal concern, but one which allows testing.

Converting the *abstract-deque* package proved a little more challenging, as the typeclass interface requires knowledge of both the queue type and the monad results are produced in. This issue was solved by use of type families. This solution is not ideal as it adds explicit knowledge of MonadConc to the DequeClass typeclass, but it suffices for testing purposes:

18 https://hackage.haskell.org/package/abstract-deque
19 https://hackage.haskell.org/package/monad-par
20 https://hackage.haskell.org/package/mwc-random
21 https://hackage.haskell.org/package/mtl

```
makeScheds :: Int -> IO [Sched]
makeScheds main = do
  caps <- getNumCapabilities
  workpools <- replicateM caps R.newQ
  rngs <- replicateM caps
            (Random.create >>= newHotVar)
  idle <- newHotVar []

  sessionFinished <- newHotVar False
  let sess = [Session baseSessionID sessionFinished]
  sessionStacks <- mapM newHotVar
                     (replicate caps sess)
  activeSessions <- newHotVar S.empty
  sessionCounter <- newHotVar (baseSessionID + 1)
  let allscheds =
       [ Sched { no=x, idle, isMain=(x==main),
                 workpool=wp, scheds=allscheds,
                 rng=rng, sessions=stck,
                 activeSessions=activeSessions,
                 sessionCounter=sessionCounter
               }
         | x    <- [0 .. caps-1]
         | wp   <- workpools
         | rng  <- rngs
         | stck <- sessionStacks
       ]
  return allscheds
```

Original

```
makeScheds :: MonadConc m => Int -> m [Sched m]
makeScheds main = do
  caps <- getNumCapabilities
  workpools <- replicateM caps R.newQ
  rngs <- replicateM caps
            (newHotVar (mkStdGen 0))
  idle <- newHotVar []

  sessionFinished <- newHotVar False
  let sess = [Session baseSessionID sessionFinished]
  sessionStacks <- mapM newHotVar
                     (replicate caps sess)
  activeSessions <- newHotVar S.empty
  sessionCounter <- newHotVar (baseSessionID + 1)
  let allscheds =
       [ Sched { no=x, idle, isMain=(x==main),
                 workpool=wp, scheds=allscheds,
                 rng=rng, sessions=stck,
                 activeSessions=activeSessions,
                 sessionCounter=sessionCounter
               }
         | x    <- [0 .. caps-1]
         | wp   <- workpools
         | rng  <- rngs
         | stck <- sessionStacks
       ]
  return allscheds
```

Déjà Fu

Figure 2: Par “direct” scheduler initialisation

```
class MonadConc (MConc d) => DequeClass d where
  type MConc d :: * -> *

  newQ :: MConc d (d elt)
  ...
```

Only 47 lines out of 342 across three modules needed change. 20 of these were in type signatures, 8 in type and data declarations, and 6 in typeclass instances. The remaining lines were, again, renaming `*IORef` functions to `*CRef` equivalents, import statements, and the addition of a type family as displayed in the code sample above.

With the constant value 'PRNG', a deadlock was discovered. It only arises after 200 queries. Given that the range of values is from 0 to the number of capabilities, and the probability is uniformly distributed, the probability of an actual deadlock is about 4×10^{-121} on a quad-core computer. No deadlocks were discovered when using the `StdGen` generator, with a variety of initial seeds tried. If there is still a deadlock, it may require more than 2 capabilities to manifest.

Due to the very contrived nature of the one deadlock found, and the length of the trace, it is not displayed here. If there is anything that can be taken away from this experiment with the Par monad, it is that it is very reliable.

7 PRACTICAL USAGE

A tool is effectively useless if it is too difficult to use. The main obstacle to the use of Déjà Fu is existing libraries which use `IO`; one cannot simply use `liftIO` everywhere, without almost certainly sacrificing completeness in all but trivial examples. Ideally, existing libraries would be modified to support the `MonadConc` abstraction. However, this does not seem a likely short-term goal, and so a more promising way to approach the problem is to provide feature-complete alternatives to existing libraries. As adapting code to `MonadConc` is not very difficult given the ability to also modify the dependencies, as seen in the Par monad case study, this is a viable short-term solution.

A second, more tractable, problem is integration with existing testing frameworks; using Déjà Fu for little stand-alone programs is all well and good, but in order to take off, it must be easy to use in the testing context people are already familiar with.

7.1 *Alternatives to Existing Libraries*

There are popular Haskell libraries specifically for concurrency. One of these is the *async*[22] library, for expressing asynchronous computations. This library is intended to be a higher-level and safer way of expressing asynchronous computations than using `forkIO` and `MVars` directly. It provides two main functions to execute an action asynchronously:

```
async :: IO a -> IO (Async a)
withAsync :: IO a -> (Async a -> IO b) -> IO b
```

22 https://hackage.haskell.org/package/async

Both of these fork the computation into a separate thread, returning an `Async` value, which contains an `MVar` which can be blocked on in order to retrieve the result. In addition, `withAsync` kills the thread if the inner action completes before it, to help prevent resource leaks.

There is a further abstraction atop `Async`, called `Concurrently`, which has Functor, Applicative, and Alternative instances, and represents an action which can be composed with other actions and execute concurrently. The concurrency is achieved by having `(<*>)` execute each action asynchronously. There was a Monad instance proposed for `Concurrently`, but this broke the laws, as `ap` was not the same as `(<*>)`[23]. This was due to `ap` executing its arguments sequentially, the only option with `(>>=)`.

This law-breaking could have been discovered through testing, but only probabilistically. If *async* were written using `MonadConc`, the relevant laws could have been specified as unit tests and checked. The bug might then have been caught before it showed up in user code.

To address both of these issues, there is an *async-dejafu* package, which provides almost the same API as *async*, but is parameterised by a `MonadConc`, giving functions like this:

```
async :: MonadConc m => m a -> m (Async m a)
withAsync :: MonadConc m => m a -> (Async m a -> m b) -> m b
```

There is a test suite using Déjà Fu. The test suite for *async* just runs most tests a single time, although one of them is run 1000 times. Using Déjà Fu here to automatically seek out interesting schedules is a much more principled approach then repetition and hope.

Not all of the features of *async* are supported by *async-dejafu*: as `MonadConc` does not support bound threads, those functions that use them have been omitted.

Of course, *async* is just one library, and providing an alternative library people will have to switch to is far from optimal. However, until library authors start to use Déjà Fu and `MonadConc` directly, such alternatives will be needed to answer the question "Why should I use this if I can't use it with all of my familiar tools?"

7.2 *Integration with Testing Frameworks*

There are two popular libraries for unit testing in Haskell, *HUnit*[24] and *tasty*[25]. From the perspective of the user, both libraries are very similar, but from the perspective of the implementer, they have different approaches to integration. Packages *hunit-dejafu*[26] and *tasty-dejafu*[27] provide integration with both.

These packages provide a common set of testing functions, an analogue of Test.DejaFu but constructing values representing individual tests which the frameworks can run, rather than executing and printing results directly:

```
testAuto    :: (Eq a, Show a) => (forall t. ConcST t a) -> Test
testDejafu  :: Show a => (forall t. ConcST t a) -> String -> Predicate a -> Test
testDejafus :: Show a => (forall t. ConcST t a) -> [(String, Predicate a)] -> Test
```

23 https://github.com/simonmar/async/pull/26
24 https://hackage.haskell.org/package/HUnit
25 https://hackage.haskell.org/package/tasty
26 https://hackage.haskell.org/package/hunit-dejafu
27 https://hackage.haskell.org/package/tasty-dejafu

Here `Test` is the type of individual tests, from *HUnit*. The *tasty* library uses `TestTree`, which has a similar purpose; it also uses `TestName` rather than `String`. To complete the set, there are variants of these functions for `ConcIO`, and also taking the schedule bounds and memory type as parameters. All of the testing functions are implemented in terms of `testDejafus'` and `testDejafusIO'`.

The *test-framework*[28] library is also in common use, however it supports integration with *HUnit*, and so needs no special support. of its own.

HUnit

Tests in *HUnit* are just thinly wrapped `IO ()` actions, which can be grouped together into collections and given names. The testing model is very simple: a test fails if and only if it produces some output. Test-running functions throw an exception if they fail, terminating the rest of the test case.

```haskell
test :: Show a => MemType -> Bounds -> (forall t. ConcST t a)
  -> [(String, Predicate a)] -> Test
test memtype cb conc tests = case map toTest tests of
  [t] -> t
  ts  -> TestList ts
  where
    toTest (name, p) = TestLabel name . TestCase . assertString . showErr $ p traces
    traces = sctBound memtype cb conc
```

Here, each `(String, Predicate a)` pair is turned into a separate test case. If there is only one, it is returned directly, otherwise the test cases are grouped together into a `TestList`.

The `assertString` function is provided by *HUnit*. The test fails if its string argument is non-empty, `showErr` here is a function to pretty-print the failures.

tasty

In contrast to the simple function-based method of *HUnit*, *tasty* has a much more complex approach. It defines a typeclass `IsTest` of things which can be converted to a unit test:

```haskell
test :: Show a => MemType -> Bounds -> (forall t. ConcST t a)
  -> [(TestName, Predicate a)] -> TestTree
test memtype cb conc tests = case map toTest tests of
  [t] -> t
  ts  -> testGroup "Deja Fu Tests" ts
  where
    toTest (name, p) = singleTest name $ ConcTest traces p
    traces = sctBound memtype cb conc
```

This is very similar to the *HUnit* approach, but instead of constructing a test value directly, it constructs an intermediate `ConcTest` value. Note also that *tasty* does not allow nameless test lists.

28 https://hackage.haskell.org/package/test-framework

```
data ConcTest where
  ConcTest :: Show a => [(Either Failure a, Trace)] -> Predicate a -> ConcTest
  deriving Typeable

instance IsTest ConcTest where
  testOptions = return []

  run _ (ConcTest traces p) _ =
    let err = showErr $ p traces
     in return $ if null err then testPassed "" else testFailed err
```

The *tasty* library is definitely more featureful than *HUnit*, but this comes at the cost of additional complexity for developers trying to integrate new functionality.

8 CONCLUSIONS, RELATED & FUTURE WORK

This work was inspired by attending a talk at INVEST 2014[29] by Paul Thompson, then a Ph.D student at Imperial College London, on systematic concurrency testing, the talk mentioned tools for languages like Java and C, but functional languages were not mentioned at all. The questions to be answered, then, were:

- **Can concurrency testing techniques from the imperative and object-oriented worlds be applied in the functional world?**

  The answer to this would seem to be a resounding "yes". There was some difficulty in implementing these techniques in a purely functional setting, as the algorithms are typically expressed in terms of mutable state, but this was overcome.

- **Does the purely functional setting allow for new techniques to be developed?**

  Initially it was hoped that the lack of side-effects in regular evaluation, amongst other things, would allow for new techniques to be developed. Unfortunately, as concurrency testing explicitly cares about *execution* rather than *evaluation* (although these are one and the same in most languages), this does not seem to be the case.

In [Walker and Runciman, 2015], we asked if the cost of a programmer needing to write their code in terms of `MonadConc` rather than `IO` was too high, and would discourage people. This is still a concern, but with the development of libraries to integrate with or replace others, we hope that the use of Déjà Fu will appear attractive enough to overcome this.

The contributions of this work are:

- a generalisation of a large subset of the GHC concurrency abstraction;
- a library called Déjà Fu for the systematic testing of concurrent Haskell programs, including those using relaxed-memory effects.

29 http://wp.doc.ic.ac.uk/verificationgroup/event/workshop-introduction-to-verification-and-testing-invest-2014/

8.1 *Related Work*

There is a tension in testing concurrent programs between *verification* and *bug-finding*. For verification, completeness is desirable, whereas for testing completeness can be sacrificed if the number of defects found in non-contrived examples is not affected much. Furthermore, by sacrificing completeness, speed can be gained, which is of great importance for developers running a test suite repeatedly as development proceeds.

Partial-order methods were first introduced in [Godefroid, 1996], which also introduced the insight that a concurrent execution can be thought of as a *partial-order* of the dependent actions in the system. Initially, these methods were based on a static analysis of the program under test. Further developments in [Flanagan and Godefroid, 2005] enabled the information needed for partial-order methods to be obtained at run-time system, often leading to a reduction in the amount of work done. The static analysis is necessarily *conservative*, whereas the dynamic analysis has much more complete information available to it.

A different approach to testing concurrent programs was explored in [Musuvathi and Qadeer, 2007], where executions exceeding some pre-determined *bound* are simply not done. Completeness is sacrificed in return for more rapid results of testing, on the assumption (later validated by empirical studies such as [Thomson et al., 2014]) that realistic test cases could find bugs within a small bound.

It was later shown in [Coons et al., 2013] that these two approaches, partial-order reduction and schedule bounding, can be unified. The result is necessarily incomplete. However it can reduce the number of executions tried to a far greater extent than either of the two component methods alone. With the evidence that schedule bounding isn't a problem in practice for testing, this became a widely-adopted method.

An assumption of key importance in concurrency testing is that all nondeterminism arises from the scheduler. Most other sources, such as random number generators, can be handled by (for example) using a fixed seed. However, in the quest for ever more performance, hardware manufacturers imposed *relaxed memory* architectures on programmers, where reads and writes done in parallel can give results impossible under sequential consistency.

[Zhang et al., 2015] showed how this additional source of nondeterminism can be handled, by modelling a single level of cache (which corresponds to total-store order or partial-store order) as simply a separate thread, committing writes to memory.

A different approach to reducing the work done as a refinement of a pure partial-order reduction approach was taken in [Sen, 2007], which uses random scheduling. Partial-order reduction is used to prune the search space, but random decisions are made where there are still multiple choices available. Random scheduling itself does not necessarily work very well, as some partial orders have more total order refinements than others, hence pruning the search space in partial-order reduction was reported to be an effective way to increase the bug-finding ability of random scheduling.

This work was furthered in [Sen, 2008], which does away with the partial-order reduction entirely in favour of a simpler race condition detection approach. The algorithm consists of two phases: firstly, all pairs of possibly-racing operations are computed; secondly, for each pair, execution proceeds with a random scheduler. When one of the identified statements is about to be executed, that thread is instead postponed until another thread is about to execute the other statement, the race is then randomly resolved and execution continues. Rather than exploring all partial orders, this approach is a probabilistic one, but is guaranteed to explore only *racing* partial orders. This approach has an advantage in programs which have many non-racy partial orders, where randomly choosing between them does not reliably produce a bug.

Pulse [Claessen et al., 2009] is a user-level scheduler for Erlang programs implementing co-operative multi-tasking. An instrumentation process automatically modifies existing programs to call out to this scheduler. Pulse works by only allowing one of the threads to operate at a time, and makes scheduling decisions around actions with side-effects: such as a process receiving a message. It also allows interaction with uninstrumented functions, which are treated as atomic, allowing tested subsystems to be composed without exploring interleavings within the subsystem. This is not possible in general in Déjà Fu due to the support for relaxed memory, which Erlang does not have. Pulse scheduling decisions are made randomly, using a provided seed, and a complete execution trace is returned, which can be rendered into a graphical form showing the interactions between threads to aid debugging. Although Pulse is not a concurrency testing tool as such, it is a core component of one, and testing can be done by simply trying different random seeds. The authors report that the graphical traces can often suggest potential race conditions which have not been evident to a human reader.

Sen's 2008 work was then used in [Arts et al., 2011] to improve race condition detection in Erlang. Pulse is used to generate an execution trace, which is then examined for possible race conditions, which are delayed and randomly resolved as in Sen's work. The authors reported that improvements can result in new bugs being found, although in the cases where the procrastination was not necessary to find the bug, performance degrades. This is because one test with procrastination is actually several program executions with different schedules.

Although the `MonadConc` typeclass was structured to be similar to the standard concurrency primitives, the inspiration for this approach, and the basic idea behind how to do SCT in Haskell, was provided by [Ankuzik, 2014]. However, both the family of primitives and the approach to testing have been significantly advanced.

An earlier version of this work was published as [Walker and Runciman, 2015]. The version discussed in this publication does not make any use of partial-order reduction, relying solely on schedule bounding to prune the search space. As a result it suffers from state explosion as programs under test become larger, and is less applicable to real-world applications. Furthermore, it lacks any support for relaxed memory. This was originally a design decision, on the assumption that most real-world concurrent Haskell programs do not use such behaviour, but initial feedback led to this decision being revised.

8.2 *Future Work*

There are a number of areas available for further exploration.

Verification of Déjà Fu Work has already begun on the formalisation in Isabelle/HOL of prefix validity in the SCT implementation. The other open issues of verification were discussed in Section 5, but to summarise, these are: correctness of primitive actions; granularity of scheduling decisions; generated schedule prefix validity; and result completeness.

Memory model for GHC Haskell / C-- In order to fully validate the testing stepwise executor, a formulation of the memory model is necessary. One way to approach this would be a formulation for all of the GHC Haskell primitives. As these are written in C--, which has no memory model, a formulation of that would also be necessary. Work on formalising the C++11 memory model in [Batty et al., 2011] may be of use here.

Generating test cases for concurrent APIs The QuickSpec tool, introduced in [Claessen et al., 2010], can generate laws that a collection of functions appear to hold based on random testing. It can be used as a way to obtain suitable test cases, if the user selects the generated laws that *should* hold, rather than those which merely *accidentally* hold. A QuickSpec-like tool which can generate laws about a concurrency-using API would be interesting and useful.

Multi-level memory caching The current approach taken for modelling relaxed memory assumes only a single level of cache. This works well for x86 processors. Other devices, such as GPUs, group cores together where each core has a cache, and each group also has a cache. This means writes can be visible to some but not all threads. A simple way to model this would be to make group assignment static, and to have more types of commit. This would require some implementation change, but is not a large difference in algorithm. The situation becomes much more complex if group assignment is *not* static, introducing yes another source of nondeterminism.

Application to distributed systems Different threads in a concurrent program need not operate on the same machine, as long as the programmer cannot detect this. The major difficulty is the possibility of *communication failure*, which cannot happen when operating on a single machine. Another is the memory model. A single level of cache corresponds roughly to a central server with all communication going through it, rather than between nodes directly. This can be alleviated with multiple levels of caching, but still results in undesirable centralisation. Work on modelling concurrent data stores as replicated eventually-consistent data types in [Burckhardt et al., 2014] may be relevant.

Appendices

A PRIMITIVE ACTIONS

There are currently 31 primitive actions used to construct the testing instances of `MonadConc`, one of which only arises when testing under relaxed memory. These primitive actions contain a continuation, allowing individual actions to be composed into larger execution sequences. Each thread of execution consists of such a sequence, terminated by the `AStop` primitive, which has no continuation and signals the termination of the thread.

Threads, `CVars`, `CRefs`, and `CTVars` may be given names, which are displayed in execution traces. If no name is given, a unique numeric identifier is used instead.

Threading

AFork *name (unmask → action) (thread_id → action)*	Create a new thread from the first action, and continue executing the current thread with the second.
AMyTId *(thread_id → action)*	Continue execution of the current thread by querying the thread identifier.
AYield *action*	Execute the given action, but also signify to the scheduler that it may be worth running a different thread now.
AStop	Terminate the current thread.

CRefs

ANewRef *name a (cref a → action)*	Construct a new `CRef` and give it to the continuation.
AReadRef *(cref a) (a → action)*	Read the currently visible value of a `CRef`.
AReadRefCas *(cref a) (ticket a → action)*	Produce a `Ticket` from the currently visible state of a `CRef`.
APeekTicket *(ticket a) (a → action)*	Get the value out of a `Ticket`.
AModRef *(cref a) (a → (a, b)) (b → action)*	Commit all pending writes and atomically modify the value within a `CRef`.
AModRefCas *(cref a) (a → (a, b)) (b → action)*	Commit all pending writes and atomically modify the value within a `CRef` using a compare-and-swap.
AWriteRef *(cref a) a action*	Update the value of a `CRef`. The updated value is visible to the current thread immediately.
ACasRef *(cref a) (ticket a) a ((succeeded?, ticket a) → action)*	Update the value of a `CRef` if it hasn't changed since the ticket was produced..
ACommit *thread_id cref_id*	Make the last write to the given `CRef` by that thread visible to all threads.

CVars

ANewVar *name (cvar a → action)*	Construct a new `CVar` and give it to the continuation.
APutVar *(cvar a) a action*	Block until the `CVar` is empty and put a value into it.
ATryPutVar *(cvar a) a (succeeded? → action)*	Try to put a value into the `CVar` without blocking.
AReadVar *(cvar a) (a → action)*	Block until the `CVar` is full and read its value.
ATakeVar *(cvar a) (a → action)*	Block until the `CVar` is full and take its value.
ATryTakeVar *(cvar a) (Maybe a → action)*	Try to take the value from a `CVar` without blocking.

Exceptions

AThrow *exception*	Raises an exception in the current thread, terminating the current execution.
AThrowTo *exception action*	Raises an exception in the other thread, blocking if the other thread has exceptions masked.
ACatching *(exception → handler) action continuation*	Registers a new exception handler for the duration of the inner action.
APopCatching *action*	Remove the exception handler from the top of the stack.
AMasking *masking_state (unmask → action) continuation*	Executes the inner action under a new masking state, and also gives it a function to reset the masking state.
AResetMask *set? inner? masking_state action*	Sets the masking state.

Software Transactional Memory

AAtom *transaction continuation*	Execute an STM transaction atomically.
SNew *name a (ctvar a → action)*	Create a new `CTVar` containing the given value.
SRead *(ctvar a) (a → action)*	Read the current value of a `CTVar`.
SWrite *(ctvar a) a action*	Update a `CTVar`.
SThrow *exception*	Throw an exception, aborting the current execution flow.
SCatch *(exception → handler) action continuation*	Registers a new exception handler for the duration of the action.
SRetry	Abort the current transaction.
SOrElse *transaction transaction continuation*	Try executing the first transaction, if it fails, execute the second.

Testing Annotations

AKnowsAbout *(Either cvar ctvar) action*	Record that the thread has access to the given variable.
AForgets *(Either cvar ctvar) action*	Record that the thread no longer has access to the given variable.
AAllKnown *action*	Record that all variables the thread knows about have been reported.

Miscellaneous

AReturn *action*	Execute the given action.
ALift *monadic_action*	Execute an action from the underlying monad.

BIBLIOGRAPHY

Ankuzik. Testing a Multithreaded Application, 2014. URL http://kukuruku.co/hub/haskell/haskell-testing-a-multithread-application.

Thomas Arts, John Hughes, Ulf Norell, Nicholas Smallbone, and Hans Svensson. Accelerating Race Condition Detection Through Procrastination. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 14–22. ACM, 2011.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66. ACM, 2011.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284. ACM, 2014.

Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 149–160. ACM, 2009.

Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *Proceedings of the 4th International Conference on Tests and Proofs*, TAP'10, pages 6–21. Springer-Verlag, 2010.

Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded Partial-order Reduction. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 833–848. ACM, 2013.

E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971. URL http://dx.doi.org/10.1007/BF00289519.

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded Scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 411–422. ACM, 2011.

Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121. ACM, 2005.

GHC Base Libraries. Control.Concurrent, 2015. URL https://hackage.haskell.org/package/base-4.8.1.0/docs/Control-Concurrent.html#g:14.

Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD thesis, 1996.

N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7): 18–41, July 1993.

Simon Marlow, Ryan Newton, and Simon Peyton Jones. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82. ACM, 2011.

Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455. ACM, 2007.

Madanlal Musuvathi and Shaz Qadeer. Fair Stateless Model Checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 362–371. ACM, 2008.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280. USENIX Association, 2008.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308. ACM, 1996.

Terry Pratchett. *Thief of Time*. HarperCollins, New York, 2001.

Koushik Sen. Effective Random Testing of Concurrent Programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 323–332. ACM, 2007.

Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21. ACM, 2008.

Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213. ACM, 1995.

Michael Snoyman. Announcing auto-update, 2014. URL http://www.yesodweb.com/blog/2014/08/announcing-auto-update.

Paul Thomson, Alastair F Donaldson, and Adam Betts. Concurrency Testing Using Schedule Bounding: an Empirical Study. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 15–28. ACM, 2014.

John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, September 1994. URL http://doi.acm.org/10.1145/187387.187391.

Michael Walker and Colin Runciman. Déjà Fu: A Concurrency Testing Library for Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 141–152. ACM, 2015.

Junfeng Yang, Heming Cui, and Jingyue Wu. Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done About It. In *Presented at the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013.

Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 250–259. ACM, 2015.