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 runtime-provided primitives, or emulated versions provided as part of a testing framework.

This report discusses the design and implementation of this library, called Déjà Fu, including some case studies and the community reception of the initial version presented at the 2015 Haskell Symposium.
# CONTENTS

1 INTRODUCTION
   1.1 Parallelism vs Concurrency ............................................. 1
   1.2 Testing Concurrent Programs .......................................... 2
   1.3 Scope ............................................................................ 2
   1.4 Goals & 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 ......................................................... 18
   3.3 Scheduling ..................................................................... 28

4 SYSTEMATIC CONCURRENCY TESTING .................................. 30
   4.1 Schedule Bounding .......................................................... 30
   4.2 Partial-order Reduction ................................................... 31

5 CORRECTNESS ..................................................................... 35
   5.1 Of Execution .................................................................... 35
   5.2 Of Testing ........................................................................ 36

6 CASE STUDIES .................................................................... 39
   6.1 auto-update ..................................................................... 39
   6.2 Search Party ................................................................... 40
   6.3 The Par Monad .................................................................. 41

7 PRACTICAL USAGE ............................................................. 47
   7.1 Integration with Testing Frameworks .................................. 47
   7.2 Alternatives to Existing Libraries ..................................... 49

8 FUTURE RESEARCH & CONCLUSIONS ................................ 51
   8.1 Community Reception ...................................................... 51
   8.2 Related Work ................................................................... 51
   8.3 Future Work ..................................................................... 53
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 even leading to death [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 runtime. 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 fixed 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 implementation detail, 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* [Jones et al., 1996]. The interleaved execution of threads, when combined with mutable state, gives rise to nondeterminism. Semaphores and locks 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, and 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 Chapter 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 technique for avoiding 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 be confident 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 (as in LazyLocks [Paul Thompson, to appear]), 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 is *complete*: it 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 which does not unavoidably require support from the runtime, and which 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 this is influenced by other non-Haskell processes accessing the same file.

• Bound threads, as these affect which operating system thread FFI calls operate on, and so alters program behaviour in a parallel setting.

• `BlockedIndefinitely` exceptions, as this is a garbage collection problem, which is out of the reach of the runtime. There are some annotation functions to record which shared state a thread knows about, and so this can be supported on a limited scale, but not even GHC makes any guarantee of it being reliable.¹

1.4 GOALS & CONTRIBUTIONS

The primary contribution of this report is a library for testing concurrent Haskell programs based on a somewhat novel algorithm for reducing the number of schedules to test.

Our contributions can be seen as follows:

• Existing results from the concurrency testing world have been ported to functional programming, an area historically without much in this topic, and verified to apply.

• A 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].

1.5 REPORT ROADMAP

Firstly we explore the problem of testing concurrent programs and how it can be done. In Chapter 2 we discuss our typeclass abstraction for concurrency and how it relates to GHC’s standard concurrency API in terms of functionality. Chapter 3 explains how, given a monadic action polymorphic in the monad (as long as it belongs to our typeclass) we can execute it with a given scheduler, and Chapter 4 extends this to cover a systematic exploration of the space of all schedules. Chapter 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 Chapter 6. Chapter 7 further discusses the usage of Déjà Fu in combination with existing code. To conclude, Chapter 8 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]
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. Care must be taken that it does not become so rich that it becomes difficult to implement, however.

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

1. There is the MonadConc typeclass, which abstracts over much of the operations provided in the Control.Concurrent hierarchy, and also some other functionality like mutable memory cells (IORefs) 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 for IO, and so existing code using only the functions generalised over can be made suitable for testing quite simply. Existing code which makes use of more functionality may require a light dusting of liftIO where it is safe to do so, which will be expanded upon in §2.1.6.

---

1 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.
2 Haskell modules are arranged into a hierarchy, corresponding to files and directories.
3 A typeclass has instances, each type may have one unique instance for a typeclass.
4 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.
2.1.1 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.

We can start a new thread with the function:\footnote{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 m` is a type constructor applied to the type variable `m`, which is constrained to be a type with an instance of `MonadConc`.}

\[fork :: MonadConc m \Rightarrow m () \rightarrow m (ThreadId m)\]

This 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`:

\[myThreadId :: MonadConc m \Rightarrow m (ThreadId m)\]

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 runtime 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:

\[getNumCapabilities :: MonadConc m \Rightarrow m Int\]
\[forkOn :: MonadConc m \Rightarrow Int \rightarrow m () \rightarrow m (ThreadId m)\]

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. 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`):

\[spawn :: MonadConc m \Rightarrow m a \rightarrow m (CVar m a)\]

This 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, where every time the runtime 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:

\[yield :: MonadConc m \Rightarrow m ()\]

This 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.
Threading and the Foreign Function Interface

In order to accommodate Foreign Function Interface (FFI) calls which may block, GHC provides a mechanism for binding 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 runtime 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 this behaviour. Unfortunately, if bound threads are required, IO will have to be used.

A few predicates are provided for compatibility:

\[
\text{rtsSupportsBoundThreads :: Bool}
\]
\[
\text{rtsSupportsBoundThreads = False}
\]

\[
\text{isCurrentThreadBound :: MonadConc m => m Bool}
\]
\[
\text{isCurrentThreadBound = return False}
\]

2.1.2 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 \text{spawn} function.

The simplest type of mutable shared state provided is the \text{CRef} (Concurrent Reference). \text{CRef}s are shared variables which can be written to and read from:

\[
\text{newCRef :: MonadConc m => a -> m (CRef m a)}
\]
\[
\text{readCRef :: MonadConc m => CRef m a -> m a}
\]
\[
\text{modifyCRef :: MonadConc m => CRef m a -> (a -> (a, b)) -> m b}
\]
\[
\text{writeCRef :: MonadConc m => CRef m a -> a -> m ()}
\]

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

\[
\text{atomicWriteCRef :: MonadConc m => CRef m a -> a -> m ()}
\]

However, synchronisation can slow down execution in a parallel environment. Note that \text{modifyCRef} is also synchronised.

2.1.3 Compare-and-swap (CAS)

As \text{CRef}s 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 \text{Ticket} here:

\[\text{Function application in Haskell uses no special syntax, only juxtaposition, so return False is applying the value False to the function return. return here is used to inject a value into a monad, it is unfortunately named and has nothing to do with the return keyword in other languages.}\]
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, is strict in the value written, and will replace 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

2.1.4 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:

ewEmptyCVar :: 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 CVars means that computations can become deadlocked. For example, deadlock occurs if every thread tries to take from the same CVar. The GHC runtime 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 which CVars it knows about. This is because GHC uses the garbage collector to solve this problem, which is out of the reach of Déjà Fu.

There are also additional functions provided in the Control.Concurrent.CVar and Control.Concurrent.CVar.Strict modules.
2.1.5 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:

\[
\text{catch} :: (\text{Exception } e, \text{MonadConc } m) \Rightarrow m a \rightarrow (e \rightarrow m a) \rightarrow m a
\]

\[
\text{throw} :: (\text{Exception } e, \text{MonadConc } m) \Rightarrow e \rightarrow m a
\]

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

departure The \text{IO catch} function can catch exceptions from pure code. This is not true in general for \text{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 \text{undefined}, which are arguably bugs.

Exceptions can be used to kill a thread:

\[
\text{throwTo} :: (\text{Exception } e, \text{MonadConc } m) \Rightarrow \text{ThreadId } m \rightarrow e \rightarrow m ()
\]

\[
\text{killThread} :: \text{MonadConc } m \Rightarrow \text{ThreadId } m \rightarrow m ()
\]

These functions block until the target thread is in an appropriate state to receive the exception.

What if we don’t want our threads to be subject to destruction in this way? A thread also has a \text{masking state}, which can be used to block exceptions from other threads. There are three masking states: \text{unmasked}, in which a thread can have exceptions thrown to it; \text{interruptible}, in which a thread can only have exceptions thrown to it if it is blocked; and \text{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:

\[
\text{mask} :: \text{MonadConc } m \Rightarrow ((\forall a. m a \rightarrow m a) \rightarrow m b) \rightarrow m b
\]

\[
\text{uninterruptibleMask} :: \text{MonadConc } m \Rightarrow ((\forall a. m a \rightarrow m a) \rightarrow m b) \rightarrow m b
\]

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:

\[
\text{forkWithUnmask} :: \text{MonadConc } m \Rightarrow ((\forall a. m a \rightarrow m a) \rightarrow m ())
\]

\[
\rightarrow m \ (\text{ThreadId } m)
\]

\[
\text{forkOnWithUnmask} :: \text{MonadConc } m \Rightarrow \text{Int} \rightarrow ((\forall a. m a \rightarrow m a) \rightarrow m ())
\]

\[
\rightarrow m \ (\text{ThreadId } m)
\]

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:

7 These functions take a higher-ranked type. Removing the \text{forall} stuff, we have \((m a \rightarrow m a) \rightarrow m b\), which is a function which takes a function as an argument and returns a result. The \text{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 \text{forall} it can be determined uniquely everywhere it is used.
The SomeException type is the top of the exception hierarchy, and so can be used to catch all exceptions.

### 2.1.6 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 (that's why the typeclass exists in the first place!). Thus, it is only safe 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 the 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.

### 2.2 Software Transactional Memory

CVars 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 CVars! This is unwieldy and prone to bugs.

**Software transactional memory** (STM) is the solution. STM uses CTVars, 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 CTVars, where a transaction may be aborted partway 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.

CTVars 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 CVar and don’t like the value it has, the transaction can be aborted, and the thread will block until any of the referenced CTVars have 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:

```haskell
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:\(^8\)

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

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

STM can also use exceptions:

```haskell
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.

There are utility functions for `CTVar`s provided in `Control.Concurrent.STM.CTVar`, and an STM equivalent of `CVar`s in `Control.Concurrent.STM.CTMVar`.

### 2.3 Memory Model

There are three memory models supported in Déjà Fu:

#### Sequential Consistency

The most intuitive model: 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 `CRefs` 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 does cause some blow-up in the number of schedules tested when unsynchronised operations are used, but as most of the concurrency primitives are synchronised this tends to be fairly contained.

---

8 Here `STMLike` is a type family, it is used to relate the `MonadConc` and `MonadSTM` typeclasses.
This chapter describes the execution of a concurrent program written using the MonadConc abstraction under test, 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, which is discussed in Chapter 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

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.

AStop

Terminate the current thread.

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

```haskell
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. 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:

```haskell
instance Functor (M n r s) where
  fmap :: (a -> b) -> M n r s a -> M n r s b
  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. AReturn here is a primitive action, a constructor of the Action data type. It also allows extracting a function from one computation, a value from another, and applying them.

```haskell
instance Applicative (M n r s) where
  pure :: a -> M n r s a
  pure x = M (\c -> AReturn (c x))
```
3.1 Primitive Actions

\[(<*>) \, :: \, M \, n \, r \, s \, (a \to b) \to M \, n \, r \, s \, a \to M \, n \, r \, s \, b\]
\[(M \, f) \, <*> \, (M \, v) \, = \, M \, (\lambda c \to f \, (\lambda g \to v \, (c \, . \, g))))\]

**AReturn action**

Execute the given action.

It’s not immediately apparent why \(\text{AReturn}\) is necessary, and why \(\lambda c \to c \, x\) can’t be used instead. This 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, \(\text{pure}\) and \(\text{return}\) must have a corresponding primitive.

Finally, the \(\text{Monad}\) instance allows sequencing of primitive actions.

\[
\text{instance } \text{Monad } (M \, n \, r \, s) \, \text{where}
\]
\[
\quad \text{return} \, :: \, a \to M \, n \, r \, s \, a
\]
\[
\quad \text{return} \, = \, \text{pure}
\]
\[
\text{instance } \text{Monad } (M \, n \, r \, s) \, \text{where}
\]
\[
\quad (\gg=) \, :: \, M \, n \, r \, s \, a \to (a \to M \, n \, r \, s \, b) \to M \, n \, r \, s \, b
\]
\[
\quad (M \, m) \, >>= \, (M \, k) \, = \, M \, (\lambda c \to m \, (\lambda x \to k \, x \, c))
\]

Each of the operations in \(\text{MonadConc}\) will now be mapped down to the primitive actions which comprise them.

### 3.1.1 Threading

All of the forking functions are implemented with the same primitive action, which corresponds most closely to \(\text{forkWithUnmask}\):\(^1\)

\[
\text{fork} \quad \text{ma} = \text{forkWithUnmask} \quad (\lambda \_ \to \text{ma})
\]
\[
\text{forkOn} \quad c \, \text{ma} = \text{forkOnWithUnmask} \, c \, (\lambda \_ \to \text{ma})
\]

The \(\text{forkWithUnmask}\) function uses the \(\text{AFork}\) primitive, which creates a new thread:

\[
\text{forkWithUnmask} \quad (M \, \text{ma}) \, = \, M \, (\text{AFork} \, \text{action}) \, \text{where}
\]
\[
\quad \text{action} \, \text{unmask} \, = \, \text{runM} \, (\text{ma} \, \text{unmask}) \, (\lambda \_ \to \text{AStop})
\]

**AFork** \((\text{unmask} \to \text{action}) \, (\text{thread_id} \to \text{action})\)

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 \(\text{forkOn}\) variant simply ignores its argument:

\[
\text{forkOnWithUnmask} \quad _ \, = \, \text{forkWithUnmask}
\]

Similarly, the \(\text{getNumCapabilities}\) function just lies:

\[
\text{getNumCapabilities} \, = \, \text{return} \, 2
\]

\(^1\) Type signatures have been omitted here as they were provided in §2.1.
In the IO implementation, these functions behave as expected. A thread can query its own identifier, by means of the AMyTId primitive:

\[
\text{myThreadId} = M \text{AMyTId}
\]

**AMyTId \( \text{threadID} \to \text{action} \)**

Continue execution of the current thread by querying the thread identifier.

A thread can signal that another should run with the AYield primitive:

\[
\text{yield} = M (\lambda c \to \text{AYield} (c ()))
\]

**AYield \text{action}**

Execute the given action, but also signify to the scheduler that it may be worth running a different thread now.

### 3.1.2 CRefs

There are two testing implementations of MonadConc, which differ in how they implement mutable state. One uses the ST monad, which allows extracting a pure result at the end, but does not allow any interaction with IO. The other uses IO. This choice of ST or IO is what determines the \(n, r, \) and \(s\) parameters of the \(M\) type.

\[
\text{newCRef} \ a = M (\text{ANewRef} \ a)
\]

**ANewRef \text{a (cref a} \to \text{action)}**

Construct a new CRef and give it to the continuation.

Internally, CRefs have a unique identifier, which is used to determine if two operations on CRefs could interfere with each other: two writes to the same CRef can, but writes to two different CRefs cannot, for example. Each CRef also contains a map from thread identifiers to values, which is used for implementing relaxed memory, allowing different threads to see values more recent than the canonical value inside the CRef.

\[
\text{readCRef} \ \text{ref} = M (\text{AReadRef} \ \text{ref})
\]

**AReadRef \text{(cref a} \to \text{action)}**

Read the currently visible value of a CRef.

\[
\text{readForCAS} \ \text{ref} = M (\text{AReadRefCas} \ \text{ref})
\]

**AReadRefCas \text{(cref a} \to \text{action)}**

Produce a Ticket from the currently visible state of a CRef.

Note that when testing under relaxed memory, AReadRef may not read the latest value, as a write by another thread may have not yet been committed; similarly AReadRefCas might not produce a ticket for the latest value either.

\[
\text{peekTicket} \ \text{tick} = M (\text{APeekTicket} \ \text{tick})
\]

**APeekTicket \text{(ticket a} \to \text{action)}**

Get the value out of a Ticket.
This will always return the value seen when the Ticket was produced, regardless of whether it has been updated since then.

\[
\text{modifyCRef} \quad \text{ref } f = \text{M} \ (\text{AModRef} \ \text{ref } f)
\]

\[
\text{modifyCRefCAS} \quad \text{ref } f = \text{M} \ (\text{AModRefCas} \ \text{ref } f)
\]

\[
\text{AModRef} \ (\text{cref } a) \ (a \rightarrow (a, b)) \ (b \rightarrow \text{action})
\]
Commit all pending writes and atomically modify the value within a CRef.

\[
\text{AModRefCas} \ (\text{cref } a) \ (a \rightarrow (a, b)) \ (b \rightarrow \text{action})
\]
Commit all pending writes and atomically modify the value within a CRef using a compare-and-swap.

\[
\text{writeCRef} \ \text{ref } a = \text{M} \ (\text{AWriteRef} \ \text{ref } a \ (c \ ()))
\]

\[
\text{casCRef} \ \text{ref } \text{tick } a = \text{M} \ (\text{ACasRef} \ \text{ref } \text{tick } a)
\]

\[
\text{AWriteRef} \ (\text{cref } a) \ a \ \text{action}
\]
Update the value of a CRef. This is visible to the current thread immediately.

\[
\text{ACasRef} \ (\text{cref } a) \ (\text{ticket } a) \ a \ ((\text{succeeded?}, \ \text{ticket } a) \rightarrow \text{action})
\]
Update the value of a CRef if it hasn’t changed since the ticket was produced.

When testing under a relaxed memory model, the AWriteRef primitive does not cause the write to be visible to threads other than the one which performed the write. An ACommit action is performed at some point later to do that.

\[
\text{ACommit} \ \text{thread_id} \ \text{cref_id}
\]
Make the last write to the given CRef by that thread visible to all threads.

The atomicWrite operation is implemented in terms of modify, rather than being given its own primitive.

\[
\text{atomicWriteCRef} \ \text{ref } a = \text{modifyCRef} \ \text{ref} \ (\text{const} \ (a, ()))
\]

3.1.3 CVars

The implementation of CVars is very similar to CRefs, the differences are that there are no relaxed memory issues to worry about, and that a CVar may be empty.

\[
\text{newEmptyCVar} = \text{M} \ \text{ANewVar}
\]

\[
\text{ANewVar} \ \text{cvar} \ a \rightarrow \text{action}
\]
Construct a new CVar and give it to the continuation.

\[
\text{putCVar} \ \text{cvar} \ a = \text{M} \ (\text{APutVar} \ \text{cvar} \ a \ (c \ ()))
\]

\[
\text{tryPutCVar} \ \text{cvar} \ a = \text{M} \ (\text{ATryPutVar} \ \text{cvar} \ a)
\]

\[
\text{APutVar} \ (\text{cvar} \ a) \ a \ \text{action}
\]
Block until the CVar is empty and put a value into it.
**ATryPut (cvar a) (succeeded? → action)**
Try to put a value into the CVar without blocking.

**readCVar cvar = M (AReadVar cvar)**

**AReadVar (cvar a) (a → action)**
Block until the CVar is full and read its value.

**takeCVar cvar = M (ATakeVar cvar)**
tryTakeCVar cvar = M (ATryTakeVar cvar)

**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.

Furthermore, all the CVar operations (other than creating a new one) cause all uncommitted CRef writes to be committed.

### 3.1.4 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, the discussion in §3.2 will elaborate.

**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.

**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 terminates 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.

**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, which is appended to with the ACatching primitive, and removed from with the APopCatching primitive, which 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 when entering a new masking state:

\[
\text{mask} \ (M \ mb) = M \ (A\text{Masking} \ \text{MaskedInterruptible} \ (f \to mb \ f))
\]

\[
\text{uninterruptibleMask} \ (M \ mb) = M \ (A\text{Masking} \ \text{MaskedUninterruptible} \ (f \to mb \ f))
\]

**AMasking masking_state (unmask \rightarrow \text{action}) continuation**

Executes the inner action under a new masking state, and also gives it a function to reset the masking state.

Similarly to ACatching, this also has a counterpart primitive to undo its effect, called AResetMask. One might wonder why this is necessary, and why AMasking can’t just be used to both set and reset the masking state, but this is so that more informative execution traces can 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.

### 3.1.5 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 for threads to interfere with each other.

\[
\text{atomically} \ \text{stm} = M \ (A\text{Atom} \ \text{stm})
\]

**AAtom transaction continuation**

Execute an STM transaction atomically.

When a transaction is executed by AAtom, one of three possible conditions are differentiated between: the transaction completed successfully, and returned a value; the transaction aborted due to an uncaught exception; and 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. There are far fewer MonadSTM operations than MonadConc ones, so the entirety of the STM implementation is presented here.
retry = S (\_ -> SRetry)

SRetry
Abort the current transaction.

the check function is provided as a simple wrapper around retry:

check b = if b then return () else retry

Transactions can be composed into larger atomic transactions, and so it is necessary to be able to handle an aborting transaction without needing to break atomicity. The orElse function allows combining transactions in this way:

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

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

throwSTM e = S (\_ -> SThrow e)

catchSTM stm handler = S (SCatch (runSTM stm) (runSTM . handler))

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.

As transactions are atomic, the handling of exceptions can be vastly simplified, where an SCatch action is performed by executing the entire inner action in one step and inspecting the result. No explicit stack of exception handlers needs to be maintained, the function call stack suffices. There are no more primitives related to exception handling than these.

newCTVar a = toSTM (SNew a)

SNew a (ctvar a → action)
Create a new CTVar containing the given value.

readCTVar ctvar = S (SRead ctvar)

SRead (ctvar a) (a → action)
Read the current value of a CTVar.

writeCTVar ctvar a = S (\c -> SWrite ctvar a (c ()))

SWrite (ctvar a) a action
Update a CTVar.

Remember that the effects of a transaction only take place when it completes successfully, so SWrite will not have any externally-visible effects until the whole thing finishes. Furthermore, executing a transaction with AAtom enforces a write barrier.
3.1.6 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:

\[
\begin{align*}
_{\text{concKnowsAbout}} \text{ var } &= M (\lambda c \rightarrow \text{AKnowsAbout } \text{ var } (c ())) \\
_{\text{concForgets}} \text{ var } &= M (\lambda c \rightarrow \text{AForgets } \text{ var } (c ())) \\
_{\text{concAllKnown}} &= M (\lambda c \rightarrow \text{AAllKnown } (c ()))
\end{align*}
\]

\textbf{AKnowsAbout (Either cvar ctvar) action}

Record that the thread has access to the given variable.

\textbf{AForgets (Either cvar ctvar) action}

Record that the thread no longer has access to the given variable.

\textbf{AAllKnown action}

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

The deadlock detection process works like so: 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.

3.1.7 Lifting from the Underlying Monad

Finally, there is the \texttt{ALift} primitive, for lifting an action from the underlying monad. This is used in the implementation of \texttt{liftIO}:

\[
\texttt{liftIO} = M (\lambda c \rightarrow \texttt{ALift } (\texttt{fmap } c \texttt{ ma}))
\]

\textbf{ALift (n action)}

Execute an action from the underlying monad.

3.2 Stepwise Execution

Each thread is represented as a sequence of primitive actions, where the continuation of one action is the next action that a thread will take.

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. In order to model all the possible effects, this step function returns a completely new set of threads and identifier source.

Normally only the chosen thread will be modified, but this allows, for example, \texttt{AFork} to create new threads, and \texttt{AThrowTo} to terminate other threads. The identifier source is used to produce fresh thread, CRef, CVar, and CTVar identifiers.

After the scheduler is consulted to choose a thread, a function is called to evaluate the next step of that thread. The simplest thing that a thread can do is to stop, which will serve as a useful example:
The simple function is defined as follows:

\[\text{simple threads'} \text{ act} = \text{return (Right (threads', idSource, act, wb))}\]

The effect of stepStop can be read as: remove the current thread from the map of live threads (the kill tid threads bit); and then return the new thread map and the name of the action to appear in the trace (Stop, here). simple is a helper function for actions which don’t modify the identifier source or have any relaxed-memory effects.

The Right\(^2\) indicates that the action completed successfully. There are a few different possible failures, such as an uncaught exception, which will terminate the current thread. If the main thread is terminated like so, the entire computation terminates with that failure.

Another simple action that a thread can perform is AReturn:

\[\text{stepReturn c} = \text{simple (goto c tid threads) Return}\]

The effect of stepReturn can be read as: extract the continuation of the action and replace the continuation of the current thread with it.

3.2.1 Threading

Threads are represented as a record type:

\[\text{data Thread n r s = Thread }\]
\[\quad \text{ (continuation :: Action n r s)}\]
\[\quad , \text{ blocking :: Maybe BlockedOn)}\]
\[\quad , \text{ handlers :: [Handler n r s]}\]
\[\quad , \text{ masking :: MaskingState)}\]
\[\quad , \text{ known :: [Either CVarId CTVarId]}\]
\[\quad , \text{ fullknown :: Bool)}\]

The continuation field contains the action to execute in the next step. blocking records whether the thread is blocked and, if so, what it is waiting for. handlers is the stack of exception handlers. masking is the masking state. known is the collection of CVars and CTVars the thread is known to have access to. allknown indicates whether _concAllKnown has been called in this thread. If allknown is True for all threads, then detection of deadlocks only involving a subset of the threads is possible.

Furthermore, a simple map is used to keep track of all the threads currently extant:

\[\text{type Threads n r s = Map ThreadId (Thread n r s)}\]

There are a number of functions to ease manipulating this structure, kill and goto are two. Another is launch, used to create a new thread:

---

2 The type Either a b type is commonly used to represent computations that might fail with an error value. By convention Left err means that the computation failed with reason err, and Right x means that the computation succeeded, producing x.
\textbf{stepFork} \ a \ b = \text{return} (\text{Right} (\text{threads}', \text{idSource}', \text{Fork newtid}, \text{wb})) \text{ where} \\
\text{threads}' = \text{goto} (b \text{ newtid}) \text{ tid} (\text{launch} \text{ tid} \text{ newtid} \text{ a} \text{ threads}) \\
(\text{idSource}', \text{newtid}) = \text{nextTId} \text{ idSource}

This is somewhat more complex than the two examples seen before, as it 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 \text{simple} cannot be used, as the identifier source is being modified.

Both \text{AMyTId} and \text{AYield} follow the simple pattern:

\textbf{stepMyTId} \ c = \text{simple} (\text{goto} (c \text{ tid}) \text{ tid} \text{ threads}) \text{ MyThreadId} \\
\textbf{stepYield} \ c = \text{simple} (\text{goto} c \text{ tid} \text{ threads}) \text{ Yield}

Note that \text{AYield} does not have any special implementation here. Its effect is purely a scheduling concern; from the point of view of updating the state of the system, it is no different to \text{AReturn}.

\section*{3.2.2 \texttt{CRef}s and Relaxed Memory}

\textbf{newtype} \texttt{CRef} \ r \ a = \texttt{CRef} (\text{CRefId}, \ r \texttt{(Map ThreadId} a, \texttt{Integer}, a))

A \texttt{CRef} is implemented as a mutable reference containing a \textit{globally visible} value, a counter of how many write commits there have been, and a number of \textit{thread-specific} values. These thread-specific values correspond to uncommitted writes, and so only show up when using relaxed memory.\footnote{\texttt{name@(pattern)} syntax is called an \textit{as-pattern}. The name before the \@ can refers to the entire value.}

\textbf{newtype} \texttt{Ticket} \ a = \texttt{Ticket} (\text{CRefId}, \texttt{Integer}, a)

A \texttt{Ticket} just keeps track of the \texttt{CRef} it was produced for, what the write count was when it was produced, and the thread-specific value seen by its creating thread. Checking if a \texttt{CRef} has been modified since the creation of a \texttt{Ticket} becomes very simple with this implementation: the write counts are compared.

\textbf{stepWriteRef} \ \text{cref@}(\text{CRef (crid, _)}) \ a \ c = \text{case} \ \text{memtype} \ \text{of} \\
\textbf{SequentialConsistency} \rightarrow \text{ do} \\
\text{writeImmediate} \ \text{cref} \ a \\
\text{simple} (\text{goto} c \text{ tid} \text{ threads}) (\text{WriteRef crid})

There are three memory models supported by Déjà Fu, each of which has a different implementation for writing to a \texttt{CRef}. Firstly, sequential consistency. This does not have any relaxed memory effects.\footnote{This is anemple of \textit{do notation}, which is a convenient syntactic sugar for composition of monadic functions.}

\texttt{SequentialConsistency} \rightarrow \text{ do} \\
\text{writeImmediate} \ \text{cref} \ a \\
\text{simple} (\text{goto} c \text{ tid} \text{ threads}) (\text{WriteRef crid})

The \texttt{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: 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. Total store order corresponds to modern x86 and x86_64 processors.

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, as this has the effect of being 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 which have this barrier property:\footnote{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.}

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 the write counts, 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:

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. This is why 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 an ACommit action:
The commitWrite function is used here. Note how the invocation 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. This is expanded upon in §3.3.

The modification is made globally visible by the use of writeImmediate. Here casCRef is used in the implementation stepModRefCas, because it is strict in the value written, which writeImmediate is not.

Reading a reference is quite simple:

The readCRef function checks if there is a cached value for that 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.

Creating a new CRef looks a little more involved, but it is really quite simple. Firstly, a new mutable reference containing the given value and no thread-specific values is created; then this is packaged up into a CRef by giving it a unique identifier; finally the thread is given it:
```
stepNewRef a c = do
  ref <- newRef (empty, 0, a)

  let (idSource', newcrid) = nextCRId idSource
  let threads' = goto (c (CRef (newcrid, ref))) tid threads

  return (Right (threads', idSource', NewRef newcrid, wb))

3.2.3 CVars

As there are no relaxed memory issues to worry about, the CVar implementation is in many respects simpler than that for CRef. However, CVars have their own unique features: specifically, blocking. Attempting to read or take from an empty CVar blocks the thread, and attempting to put into a full CVar does the same.

Firstly, creating a new CVar is almost identical to creating a new CRef:
```
stepNewVar c = do
  ref <- newRef Nothing

  let (idSource', newcvid) = nextCVId idSource
  let threads' = knows [Left newcvid] tid
    (goto (c (CVar (newcvid, ref))) tid threads)

  return (Right (threads', idSource', New newcvid, wb))

There is a difference, however. The knows function is used to record that a thread has a reference to a CVar or CTVar, which can be used to improve deadlock detection.

Fortunately there is a lot of similarity between the CVar functions, which makes them easy to follow.
```
stepPutVar cvar@(CVar (cvid, _)) a c = synchronised $ do
  (success, threads', woken) <- putIntoCVar cvar a c tid threads
  simple threads' $ if success then Put cvid woken else BlockedPut cvid

stepTryPutVar cvar@(CVar (cvid, _)) a c = synchronised $ do
  (success, threads', woken) <- tryPutIntoCVar cvar a c tid threads
  simple threads' (TryPut cvid success woken)

Note that these functions are all synchronised, and so commit CRef writes. All CVar actions (other than ANewVar) are.
stepReadVar cvar@(CVar (cvid, _)) c = synchronised $ do
  (success, threads', _) <- readFromCVar cvar c tid threads
  simple threads' $ if success then Read cvid else BlockedRead cvid

stepTakeVar cvar@(CVar (cvid, _)) c = synchronised $ do
  (success, threads', woken) <- takeFromCVar cvar c tid threads
  simple threads' $ if success then Take cvid woken else BlockedTake cvid

stepTryTakeVar cvar@(CVar (cvid, _)) c = synchronised $ do
  (success, threads', woken) <- tryTakeFromCVar cvar c tid threads
  simple threads' (TryTake cvid success woken)

The putInto/readFrom/takeFromCVar functions, and their try variants, handle waking threads which are blocked in the appropriate way on that CVar. All such threads get woken at once, and a list of them is returned to be included in the execution trace. This is somewhat different to how GHC does things, where threads blocked on an MVar are woken up in a FIFO order to guarantee fairness. The current behaviour was chosen because there is no standard for Haskell concurrency, and so that ordering is only an implementation detail which could, conceivably, be changed in the future if another were judged more desirable.

3.2.4 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 AThrowTo.

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)

This introduces the APopCatching action, at the end of both the enclosed action, and at the end of the handler. This is necessary because actions are executed one at a time, and so we cannot just run the entire inner computation in one go and then check the result for an uncaught exception.

stepPopCatching a = simple threads' PopCatching where
  threads' = goto a tid (uncatching tid threads)

The catching and uncatching functions are used to modify the handler stack, corresponding to push and pop operations.

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:

```haskell
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 it’s 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 thread is killed and is the main thread, the entire computation terminates.

```haskell
stepMasking m ma c = simple threads' (SetMasking False m) where
  a = runCont (ma umask) (AResetMask False False m' . c)

  m' = masking (lookup tid threads)
  umask mb = do
    resetMask True m'
    b <- mb
    resetMask False m
    return b
  resetMask typ ms = cont (
    k -> AResetMask typ True ms (k ())))

  threads' = goto a tid (mask m tid threads)

Similarly to ACatching, AMasking introduces an additional action into its continuation: AResetMask, which returns the masking state to what it originally was. It also constructs a function to execute an action with the original masking state, the umask function.

```haskell
stepResetMask b1 b2 m c = simple threads' action where
  action = (if b1 then SetMasking else ResetMasking) b2 m
  threads' = goto c tid (mask m tid threads)
```

3.2.5 Software Transactional Memory

As STM transactions are atomic, the implementation is vastly simplified. They are still implemented in terms of a step function, but it is just iterated until termination.
Firstly, the transaction is executed:

```haskell
stepAtom stm c = synchronised $ do
  (res, newctvid) <- runstm stm (nextCTVId idSource)
  let idSource' = idSource { nextCTVId = newctvid }
  case res of
    Success readen written val
      let (threads', woken) = wake (OnCTVar written) threads
        in return (Right (goto (c val) tid threads', idSource', STM woken, wb))
    Retry touched ->
      let threads' = block (OnCTVar touched) tid threads
        in return (Right (threads', idSource, BlockedSTM, wb))
    Exception e -> stepThrow e
```

There are now three possible results: the transaction succeeded; the transaction aborted due to calling `retry`; or the transaction aborted due to an uncaught exception.

If the transaction succeeds, all threads blocked on CTVars which were modified are woken:

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

If the transaction aborts due to `retry`, the thread is blocked until any of the read CTVars are modified:

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

If the transaction aborts due to an uncaught exception, the exception is thrown in the thread:

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

There are 9 primitive actions used to implement STM transactions, discussed in §3.1. The implementation of two of them, SNew and SLift, are virtually identical to similar primitives discussed elsewhere, and so will not be discussed here. The remaining primitives are SRead, SWrite, SCatch, SOrElse, SRetry, SThrow, and SStop.

```haskell
stepRead (CTVar (ctvid, ref)) c = do
  val <- readRef ref
  return (c val, nothing, [ctvid], [])
```

The most obvious difference is that there is only one thread of control. Moreover, a new transaction is built up to undo what has already been done (nothing in this case), so that a transaction can be reverted. Furthermore, lists of the CTVars read from and written to are constructed.

There are no relaxed memory effects in STM transactions, so reading and writing is incredibly simple.

```haskell
stepWrite (CTVar (ctvid, ref)) a c = do
  old <- readRef ref
  writeRef ref a
  return (c, writeRef ref old, [], [ctvid])
```

Here we see the inverse transaction being built up: to undo a write, write the old value. The handling of exceptions is vastly simplified, as SCatch can just execute the entire inner transaction and examine the result:
stepCatch h stm c = onFailure stm c
\(\text{\textbackslash readen} \rightarrow \text{return (SRetry, nothing, readen, [])})\)
\(\text{\textbackslash exc} \rightarrow \text{case fromException exc of}\)
\(\text{Just exc'} \rightarrow \text{transaction (h exc') c}\)
\(\text{Nothing} \rightarrow \text{return (SThrow exc, nothing, [], [])})\)

Here onFailure and transaction are functions to do different things for the three different possible results of a transaction:

transaction stm onSuccess = onFailure stm onSuccess
\(\\text{\textbackslash readen} \rightarrow \text{return (SRetry, nothing, readen, []})\)
\(\text{\textbackslash exc} \rightarrow \text{return (SThrow exc, nothing, [], [])})\)

onFailure stm onSuccess onRetry onException = do
(res, undo) <- doTransaction stm
case res of
\(\text{Success readen written val} \rightarrow \text{return (onSuccess val, undo, readen, written})\)
\(\text{Retry readen} \rightarrow \text{onRetry readen}\)
\(\text{Exception exc} \rightarrow \text{onException exc}\)

The effect of stepCatch, then, is to run the entire inner transaction. If it throws an exception, and the exception is of the right type for the handler, the handler is executed. If the handler throws an exception, it is not dealt with. If the exception is not of the right type for the handler, it propagates upwards.

The implementation of SOrElse is actually quite similar to SCatch: it runs the entire first transaction and, if it aborts due to a retry, runs the second:

stepOrElse a b c = onFailure a c
\(\text{\textbackslash _} \rightarrow \text{transaction b c})\)
\(\text{\textbackslash exc} \rightarrow \text{return (SThrow exc, nothing, [], [])})\)

The terminating cases are implemented very simply:

stepRetry = return (SRetry, nothing, [], [])
stepThrow e = return (SThrow e, nothing, [], [])
stepStop = return (SStop, nothing, [], [])

Termination is achieved by checking if the next action a thread will perform is one of these three.

3.2.6 Testing Annotations

Normally Déjà Fu can only detect a deadlock when every thread is blocked. However, it may be the case that a smaller collection of threads are deadlocked, if they are all blocked on a CVar which no thread outside the collection has a reference to, for example. GHC can do this sort of deadlock detection using its garbage collector, and can signal to threads when they are blocked.
Déjà Fu does not have access to the garbage collector, and so relies on programmer-provided hints about which CVars and CTVars are known about by which threads.

\[
\begin{align*}
\text{stepKnowsAbout} & \ v \ c = \text{simple} \ (\text{knows} \ [v] \ \text{tid} \ (\text{goto} \ c \ \text{tid} \ \text{threads})) \quad \text{KnowsAbout} \\
\text{stepForgets} & \ v \ c = \text{simple} \ (\text{forgets} [v] \ \text{tid} \ (\text{goto} \ c \ \text{tid} \ \text{threads})) \quad \text{Forgets} \\
\text{stepAllKnown} & \ c = \text{simple} \ (\text{fullknown} \ \text{tid} \ (\text{goto} \ c \ \text{tid} \ \text{threads})) \quad \text{AllKnown}
\end{align*}
\]

The \text{knows} and \text{forgets} functions are used to modify the set of variables that a thread is known to have a reference to. The \text{fullknown} function indicates that this set is complete.

If every thread is in a fully-known state, then the deadlock detection algorithm is enhanced to: for a given thread blocked on a CVar or CTVar, if no other thread which is not also blocked on the same thing has a reference to that variable, then the thread is deadlocked.

3.2.7 Lifting from the Underlying Monad

Because all of the step functions are defined in terms of the underlying monad, lifting an action is incredibly simple:

\[
\begin{align*}
\text{stepLift} \ na & = \text{do} \\
& a \gets \text{na} \\
& \text{simple} \ (\text{goto} \ a \ \text{tid} \ \text{threads}) \quad \text{Lift}
\end{align*}
\]

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. This 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 is the basis for the systematic concurrency testing implementation.

\[
\text{type} \quad \text{Scheduler} \quad s = s \\
\quad \rightarrow \quad \text{Maybe} \ (\text{ThreadId}, \ \text{ThreadAction}) \\
\quad \rightarrow \quad \text{NonEmpty} \ (\text{ThreadId}, \ \text{NonEmpty} \ \text{Lookahead}) \\
\quad \rightarrow \quad (\text{ThreadId}, \ s)
\]

In order to make nontrivial decisions, a scheduler maintains some state, of type \( s \). This could be, for example, a random number generator:

\[
\begin{align*}
\text{randomSched} & : : \text{RandomGen} \ g \Rightarrow \text{Scheduler} : g \\
\text{randomSched} \ g \_ \ \text{threads} & = (\text{threads'} \ ?? \ \text{choice}, \ g') \ \text{where} \\
& (\text{choice}, \ g') = \text{randomR} \ (0, \ \text{length} \ \text{threads'} - 1) \ g \\
& \text{threads'} = \text{map} \ \text{fst} \ \text{(toList} \ \text{threads})
\end{align*}
\]

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 = (0, ())
roundRobinSched _ (Just (prior, _)) threads
  | prior >= maximum threads' = (minimum threads', ())
  | otherwise = (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,\(^6\) 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 been done, and 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.

### 3.3.1 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 nondeterministic committing 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 do not exist in the same way that other threads do, they 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. This approach was suggested by [Zhang et al., 2015].

This approach also ensures that ACommit actions are never introduced under a sequentially-consistent memory model.

---

\(^6\) And toList converts a NonEmpty a to a [a].
SYSTEMATIC CONCURRENCY TESTING

Once the scheduling behaviour of a program can be directed at will, there is the ability to implement systematic testing. 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, there are techniques which are complete, in that they find all possible results a program could produce; and incomplete, which do not make such a guarantee.

SCT works by providing an initial sequence of scheduling decisions intended to put the program into a new state. After this point some deterministic scheduler is used, and the final trace 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 more unique initial sequences possible.

4.1 SCHEDULE BOUNDING

Schedule bounding is an incomplete approach to SCT. Each sequence of scheduling decisions is associated with a bound value, by 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.

There are some 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:

```haskell
lock p var = spin where
  spin = do
    x <- readCRef var
    unless (p x) (yield >> spin)
```
Here, a CRef is read from and, if 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 pre-emption bound of 2 and a fair bound of 5, in order to gracefully handle computations which use spinlocking techniques. The pre-emption bound was chosen based on empirical evidence, but the fair bound was picked fairly arbitrarily.

### 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 \( T \) be the set of transitions in a concurrent system. A binary, reflexive, and symmetric relation \( D \subseteq T \times T \) is a valid dependency relation iff, for all \( t_1, t_2 \in T \), \((t_1, t_2) \notin D\) (\(t_1\) and \(t_2\) are independent) implies that the 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 instead a collection of conditions sufficient for dependency are identified. These conditions are determined by what sorts of things the concurrent system under test can express.

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

\[
x \leftrightarrow y \iff \text{thread_id}(x) = \text{thread_id}(y) \lor \\
\text{variable}(x) = \text{variable}(y) \land (\text{is_write}(x) \lor \text{is_write}(y))
\]

Where \( x \leftrightarrow y \) 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:

\[
\text{dependent} \ (t_1, a_1) \ (t_2, a_2) = \text{case} \ (a_1, a_2) \ of
\]

\[
\begin{align*}
(Lift, Lift) & \quad \rightarrow \ True \\
(\text{ThrowTo} \ t, _) & \quad \rightarrow \ t == t_2 \\
(_, \text{ThrowTo} \ t) & \quad \rightarrow \ t == t_1 \\
(\text{STM} \ _, \ \text{STM} \ _) & \quad \rightarrow \ True
\end{align*}
\]

- Two lifts from the underlying monad are always dependent, as in general this allows 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 modify, 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, and do not have any relaxed memory properties. Un synchronised actions do not have this property.

\[
\text{dependentActions} \ \text{memtype} \ \text{buf} \ a_1 \ a_2 = \text{case} \ (a_1, a_2) \ of
\]

\[
\begin{align*}
(\text{UnsynchronisedRead} \ r_1, \text{UnsynchronisedWrite} \ r_2) & \quad \rightarrow \ r_1 == r_2 \\
(\text{UnsynchronisedWrite} \ r_1, \text{UnsynchronisedRead} \ r_2) & \quad \rightarrow \ r_1 == r_2 \\
(\text{UnsynchronisedWrite} \ r_1, \text{UnsynchronisedWrite} \ r_2) & \quad \rightarrow \ r_1 == r_2 \\
(\text{UnsynchronisedRead} \ r_1, _) & \quad | \ isBarrier \ a_2 \rightarrow \ isBuffered \ \text{buf} \ r_1 \\
(_, \text{UnsynchronisedRead} \ r_2) & \quad | \ isBarrier \ a_1 \rightarrow \ isBuffered \ \text{buf} \ r_2
\end{align*}
\]

\[
_ \quad \rightarrow \ \text{same crefOf} \ && (\text{isSynchronised} \ a_1 \ || \ \text{isSynchronised} \ a_2) \ || \ \text{same cvarOf}
\]

- 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 is dependent with an action that imposes a memory barrier if there are buffered writes to the variable being read from.

- 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 graph structure 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:

```haskell
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 backtracking 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`, which means that there are no unexplored backtracking points left.
4.2.1 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.

4.2.2 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 it 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.
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 OF EXECUTION

Correctness of execution asks whether the result of an arbitrary execution of Déjà Fu’s testing implementation happen 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.

5.1.1 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 ties 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, this restricts 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 in order to formally establish the memory model for CRefs, the compilation of IORefs 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.

5.1.2 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. There are executions involving the pre-emption of the evaluation of non-terminating expressions which are possible under GHC but not under Déjà Fu, but whether this can be used to produce different outputs is less clear.

5.2 OF 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.

5.2.1 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:

```haskell
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. `action` is a `Maybe` value, 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:

```haskell
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: `todo ⊆ runnable`
- Every decision in the done map is possible: `dom done ⊆ runnable`
- No decision that has been done is in the to-do set: `todo ∩ dom done = ∅`
- These properties hold recursively: `∀p ∈ ran done. 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 an important property to verify, as if they are broken any assumptions made in the rest of the code cannot be trusted.
5.2.2 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. Therefore, there are two different notions of completeness which are of interest:

• If the bounds are all set to $\infty$, 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: that 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.
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 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 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, demanding one be computed and blocking until it has been done if there isn’t one.

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 transformation to the MonadConc typeclass is mostly simple, however the threadDelay must be wrapped inside a call to liftIO. The first race condition can be exhibited with the following test:

```haskell
test :: (MonadConc m, MonadIO m) => m ()
test = do
    auto <- mkAutoUpdate defaultUpdateSettings
    auto
```

The output is as we would expect, knowing the bug is present:

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

1 https://hackage.haskell.org/package/auto-update
2 https://hackage.haskell.org/package/monad-par
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.

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

```haskell
test :: (MonadConc m, MonadIO m) => m Int

test = do
  var <- newCRef 0
  auto <- mkAutoUpdate $ defaultUpdateSettings
    { updateAction = modifyCRef var (\x -> (x+1, x)) }

  auto

  auto
```

Here `auto` is called twice to update the counter variable twice. Actually reproducing this bug requires a new `readCVar` function be written, as the library does not currently provide an option for non-atomic reads. Exhibiting this bug requires three pre-emptions:

```haskell
> dejafuIO' TotalStoreOrder 3 5 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---
```

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](https://github.com/barrucadu/search-party) 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. Initially, only single results could be returned, but support for returning all results was later added, incorrectly, introducing a bug.

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

```haskell
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 small tests in Search Party, verifying that deadlocks and exceptions don’t arise, and that results are as expected. Upon introducing this new functionality, tests began to fail with differing result lists returned for different schedules, prompting the test:

\[
\text{checkResultLists :: Eq } a \Rightarrow \text{Predicate } [a]
\]
\[
\text{checkResultLists } = \text{representative (alwaysTrue2 check) where}
\]
\[
\text{check (Right (Just as)) (Right (Just bs)) } = \text{as 'elem' permutations bs}
\]
\[
\text{check a b } = a == b
\]

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

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

[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 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 IVars are futures, not mutable state. Par uses a work-stealing scheduler running on multiple operating system threads, fully evaluating values on

---

4 The representative function picks only one trace for each unique result.
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:

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

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 [Various, 2014], 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 package and two from the monad-par 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:

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

Some simplifications were made in the conversion process:

- Par normally uses the mwc-random package when performing its internal scheduling. This was initially replaced with a constant function, and then a StdGen.

- 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.

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:

```haskell
class MonadConc (MConc d) => DequeClass d where
type MConc d :: * -> *
  newQ :: MConc d (d elt)
  ...
```

5 https://hackage.haskell.org/package/abstract-deque
6 https://hackage.haskell.org/package/monad-par
7 https://hackage.haskell.org/package/mwc-random
This solution is not ideal as it adds explicit knowledge of MonadConc to the DequeClass typeclass, but it suffices for testing purposes.

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 $4 \times 10^{-121}$ 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.
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
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

Figure 2: Par “direct” scheduler initialisation
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.

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 Integration with Testing Frameworks

There are two popular libraries for unit testing in Haskell, HUnit\(^1\) and tasty\(^2\). 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 providing integration with both, called hunit-dejafu and tasty-dejafu are provided.

Both 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:

\[
\text{testAuto} :: (\text{Eq } a, \text{Show } a) \Rightarrow (\forall t. \text{ConcST } t \ a) \rightarrow \text{Test} \\
\text{testDejafu} :: \text{Show } a \Rightarrow (\forall t. \text{ConcST } t \ a) \rightarrow \text{String} \rightarrow \text{Predicate } a \rightarrow \text{Test} \\
\text{testDejafus} :: \text{Show } a \Rightarrow (\forall t. \text{ConcST } t \ a) \rightarrow [(\text{String}, \text{Predicate } a)] \rightarrow \text{Test}
\]

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

The test-framework\(^3\) library is also in common use, however it supports integration with HUnit, and so needs no special support.

7.1.1 HUnit

Tests in HUnit are just a thin wrapper around an IO () action, 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. There are a number of provided testing functions, which throw an exception if they fail, terminating the rest of the test case.

\[
\text{test} :: \text{Show } a \Rightarrow \text{MemType} \rightarrow \text{Bounds} \rightarrow (\forall t. \text{ConcST } t \ a) \\
\quad \rightarrow [(\text{String}, \text{Predicate } a)] \rightarrow \text{Test}
\]

1 https://hackage.haskell.org/package/HUnit
2 https://hackage.haskell.org/package/tasty
3 https://hackage.haskell.org/package/test-framework
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 they are grouped together into a TestList. A TestList can consist of only one entry, but making this distinction results in a closer correspondence between the generated Test and the call to the testing function which produced it.

The assertString function is provided by HUnit. If the provided string is non-empty (showErr here is a function to pretty-print the failures, if any) the test fails.

7.1.2 tasty

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

test :: Show a => MemType -> Bounds -> (forall t. ConcST t a)
       -> [(TestName, Predicate a)] -> TestTree

where
  toTest (name, p) = singleTest name $ ConcTest traces p

  traces = sctBound memtype cb conc

This is very similar to the HUnit approach, however instead of constructing a test value directly, it constructs an intermediate ConcTest value. Note also that tasty does not allow nameless test lists. The singleTest function takes a value which is a member of the IsTest typeclass, and uses that to construct a TestTree:

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
tasty allows for passing tests to also have output associated with them, through the testPassed function, which is only displayed if the tests are executed with sufficient verbosity. Furthermore, tests can have options associated with them, which can be set when the test is executed. Neither of these features are used, as this was largely just a port of hunit-dejafu. tasty is definitely a more featureful library than HUnit, but this comes at the cost of additional complexity for developers trying to integrate new functionality.

7.2 ALTERNATIVES TO EXISTING LIBRARIES

There are a number of popular Haskell libraries specifically for concurrency. One of these is the async 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:

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

Both of these fork the computation into a separate thread, providing this Async value, containing an MVar which can be blocked on in order to retrieve the value. 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 for Concurrently, but this broke the laws, as ap was not the same as (<>*)\(^5\). This was due to ap executing its arguments sequentially, as that is all which can be done with (>>=).

This bug 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 and the bug could have been caught before it showed up in user code. Furthermore, by using IO directly, it is not possible to write a generic MonadConc action which makes use of async, which is very unfortunate.

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:

```haskell
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 D´ej`a Fu, whereas the test suite for async just runs most tests a single time, although one of them is run 1000 times. Using D´ej`a Fu here to automatically seek out interesting schedules is a much more principled approach.

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 D´ej`a Fu and

---

4 https://hackage.haskell.org/package/async
5 https://github.com/simonmar/async/pull/26
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?”
FUTURE RESEARCH & CONCLUSIONS

This work was inspired by attending a talk 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?**
  
The answer to this would seem to be a resounding “yes”. There was some difficulty in implementing these techniques to 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 definitely a concern, but with the development of libraries to integrate with or replace entirely others, we hope that the use of Dējā 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 Dējā Fu for the systematic testing of concurrent Haskell programs, including those using relaxed-memory effects.

8.1 COMMUNITY RECEPTION

A talk about Dējā Fu was given at the 2015 Haskell Symposium. The response was generally positive and the relaxed memory work was initiated following discussions about how to integrate Dējā Fu with libraries like monad-par and lvish.

Whilst no packages use Dējā Fu yet, the released version is quite primitive compared to the current developmental version, and it is hoped that the next release (and announcement) will inspire some interest.

8.2 RELATED WORK

There is a tension in testing concurrent programs between verification and bug-finding. For the former, completeness is desirable, whereas for the latter completeness can be sacrificed if the

---

1 https://hackage.haskell.org/package/monad-par
2 https://hackage.haskell.org/package/lvish
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 as they develop.

Partial-order methods were first introduced in [Godefroid et al., 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] discuss how the information needed for partial-order methods can be obtained at runtime, and shows that this often leads to a reduction in the amount of work done. This is because the static analysis is necessarily conservative, whereas the dynamic analysis has much more complete information available to it.

Meanwhile, a different approach to testing concurrent programs was being explored in [Musuvathi and Qadeer, 2007], where executions exceeding some pre-determined bound were simply not done. Completeness was sacrificed in return for more rapid results of testing, on the assumption that (later to be validated by empirical studies such as [Thomson et al., 2014]) that test cases could be written in such a way that this wasn’t a problem.

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 an enticing 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 controlled for 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 under 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 corresponding total orders than others, hence pruning the search space like this is 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 only explore 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.
8.3 future work

There are a number of areas available for further exploration.

- **Verification of Déjà Fu**
  Work has already begun on one aspect of this, the formalisation in Isabelle/HOL of prefix validity in the SCT implementation. The other open issues of verification are discussed further in Chapter 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 formalism of the memory model of the primitives used is necessary. One way to approach this would be a formalism for all of the GHC Haskell primitives. As these are written in C--, which has no memory model, a formalism 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 easily generate test cases, if the user filters the output to laws that should hold, rather than those which merely accidentally hold.

Given that concurrent programs are now easily testable, some 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, but not for other devices, such as GPUs. 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 however, as this then introduces another source of nondeterminism.

• **Application to distributed systems**

  There is no reason why different threads in a concurrent program need to operate on the same physical 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.
BIBLIOGRAPHY


