# Déjà Fu: A Concurrency Testing Library for Haskell

Michael Walker
University of York, UK
msw504@york.ac.uk

Colin Runciman
University of York, UK
colin.runciman@york.ac.uk

## Abstract

Systematic concurrency testing (SCT) is an approach to testing potentially nondeterministic concurrent programs. SCT avoids potentially unrepeatable results that may arise from unit testing concurrent programs. It seems to have received little attention from Haskell programmers. This paper introduces a generalisation of Haskell's concurrency abstraction in the form of typeclasses, and a library for testing concurrent programs. A number of examples are provided, some of which come from pre-existing packages.

***Categories and Subject Descriptors*** D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools

***Keywords*** Concurrency, functional programming, Haskell, nondeterminism, systematic concurrency testing

## 1. Introduction

Haskell has been extended in different ways to express parallelism[1] and concurrency[2]. *Deterministic* parallelism implementations, such as the Par monad[12] and Strategies[11] are very suitable for data-parallel tasks where the difficulty is efficiently performing some pure computation. However, they achieve their determinism by constraining the functionality of shared state, and so lose some convenience and utility.

In this paper, the interest is more general: computations with constrained *nondeterminism*, or interaction with the real world, as a key part of their functioning. Though very general, such computations can give rise to race conditions and deadlocks.

***Example***

```
main :: IO ()
main = do
  shared <- newMVar 0
  forkIO . void $ swapMVar shared 1
  forkIO . void $ swapMVar shared 2
  readMVar shared >>= print
```

This program may look like it will output either "1" or "2" depending on the ordering of swaps. Actually, a parent thread does not wait for child threads before terminating[10], so the output may also be "0", although this behaviour can only be exhibited in GHC with multiple OS threads. As we shall see in §4, we can test this program to produce the following trace:

```
[pass] Never Deadlocks (checked: 23)
[pass] No Exceptions (checked: 23)
[fail] Consistent Result (checked: 1)
  0 S0-------
  2 S0----P2---S0---
  1 S0----P1---S0---
```

Here each of the last three lines represents a possible execution of the program: "S" indicates the start of execution of a thread, "P" indicates the pre-emption of the running thread by another, and each dash represents one execution step.

### 1.1 Contributions

The contributions of this paper are:

- a generalisation of the standard concurrency abstraction, allowing different concrete implementations to be used;
- a library called Déjà Fu[3] for systematically testing concurrent Haskell programs for possible deadlocks, race-conditions, or uncaught exceptions, with the testing functionality based on *monadic concurrency*[14] and *schedule bounding*[13].

### 1.2 Roadmap

The rest of the paper is organised as follows.

- §2 reviews the solutions for deterministic parallelism in Haskell, and highlights their limitations.
- §3 presents the typeclass concurrency-abstraction, emphasising the few points where it departs from the usual concurrency model.
- §4 introduces writing tests with Déjà Fu.
- §5 explains how the library implements systematic concurrency testing.
- §6 presents a small selection of case studies.
- §7 gives pointers to existing concurrency testing work in both Haskell and other programming languages.
- §8 draws conclusions and suggests further work.

[1] Doing multiple things at once.

[2] Programming as if doing multiple things at once.

[3] [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)



*Haskell'15*, September 3-4, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3808-0/15/09...
http://dx.doi.org/10.1145/2804302.2804306

## 2. Deterministic Parallelism and Concurrency in Haskell

### 2.1 The Eval Monad

The Eval monad, and the Strategies[11] package built on top of it, is a way of evaluating data structures in parallel. The programmer is provided with primitives to evaluate something sequentially, in parallel (without blocking), and to run an Eval computation.

The following example[10] applies a function to both elements of a tuple in parallel, waits until the evaluation is complete, and returns the result:

```
tupleMap :: (a -> b) -> (a, a) -> (b, b)
tupleMap f (x, y) = runEval $ do
  x' <- rpar $ f x
  y' <- rpar $ f y
  rseq x'
  rseq y'
  return (x', y')
```

There is no notion of threading, or of shared mutable state. The Eval monad is for the use-case of having a large data structure which is expensive to compute.

### 2.2 The Par Monad

The Par monad[12] 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 their own threads before inserting them into an IVar. Despite its limitations, the Par monad can be very effective in speeding up pure code.

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

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

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.

### 2.3 LVish

A recent development is the LVish[9] library, which overcomes some of the limitations of the Par monad by allowing multiple writes, as long as they only ever add information to the shared structure. To maintain determinism, reads *of the same kind* return the same value that they always did. LVish allows shared state which forms a lattice, and reads correspond to seeing a small part of that lattice.

Necessarily, reads block until there is enough information in the structure to determine the result. This approach means that the entire data structure must be kept around for as long as there is a single reference, even if that reference is only ever used to read a small part of the data.

## 3. Déjà Fu: Concurrency and Haskell Revisited

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

***Departure*** Departures from the semantics of the traditional concurrency abstraction are highlighted like this. □

If we remove the limitations, allowing non-blocking reads and multiple writes, we get to Haskell's traditional concurrency abstraction in the IO monad. Déjà Fu[1] generalises a very large subset of that abstraction to work in arbitrary members of a typeclass, named MonadConc. There is an instance of MonadConc 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`s.

To make use of the Déjà Fu library, we must first import the class:

```
import Control.Monad.Conc.Class
```

### 3.1 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 `fork`[2] function:

```
fork :: ... => m () -> 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.

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 :: ... => m Int
forkOn :: ... => Int -> m () -> m (ThreadId m)
```

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

***Departure*** `getNumCapabilities` is not required to return a true result. The testing instances return "2" despite executing everything in the same capability, to encourage more concurrency. 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 :: ... => m a -> 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.

[1] `https://github.com/barrucadu/dejafu`

[2] To save on horizontal space, a `MonadConc m =>` has been omitted from type signatures.

### 3.2 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 `spawn` function.

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

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

***Departure*** IORef actions can be re-ordered[7], but this is not the case for CRef actions. The `modifyCRef` function corresponds to `atomicModifyIORef`, and `writeCRef` corresponds to `atomicWriteIORef`. □

As *any* thread can write at *any* time, we risk threads overwriting each other's work! At least `modifyCRef` is atomic: no thread can update it between the value being read and the new value being stored, as could happen if `readCRef` and `writeCRef` were composed. Even so, CRefs quickly fall down if we want to do anything complicated. We need something more robust.

### 3.3 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:

```
putCVar     :: ... => CVar m a -> a -> m ()
tryPutCVar  :: ... => CVar m a -> a -> m Bool
readCVar    :: ... => CVar m a -> m a
takeCVar    :: ... => CVar m a -> m a
tryTakeCVar :: ... => 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 (and will complain if it does), and so can Déjà Fu in a more informative way, as we shall see in §4.

### 3.4 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 part-way through depending on their values. If the transaction fails, *none of its effects take place*, and the thread blocks until the transaction can succeed. This means we need to limit the possible actions in an STM transaction, so we have another typeclass:

```
import Control.Monad.STM.Class
```

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 CTVar 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:

```
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!

We have emphasised that STM transactions are atomic. The function which atomically executes a transaction:

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

***Departure*** Every MonadConc has an associated MonadSTM, whereas there is just one STM normally. This is so that STM transactions can be tested without needing to bring IO into the test runner. The IO MonadConc instance uses STM as its MonadSTM. □

For example, suppose we have a collection of worker threads each of which can either produce a result or fail. We might want to block until either *one* completes successfully and kill the other threads, or until *all* fail. We can implement this using CTMVars, an analogue of CVars built from CTVars:

```
awaitResult :: MonadConc m
    => [(ThreadId m, CTMVar (STMLike m) (Maybe a))]
    -> m (Maybe a)
awaitResult workers = do
  out <- atomically $ do
    progress <- mapM (tryReadCTMVar . snd) workers
    let finished = catMaybes progress
    case catMaybes finished of
      (x:_) -> return $ Just x
      [] -> check (length finished < length workers)
            >> return Nothing
  mapM_ (killThread . fst) workers
  return out
```

Here `tryReadCTMVar` attempts to read from a CTMVar and, if there is no value present, calls `retry`.

### 3.5 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:

```
catch :: ... => m a -> (e -> m a) -> m a
throw :: ... => e -> m a
```

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

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

Exceptions can be used to kill a thread:

```
throwTo    :: ... => ThreadId m -> e -> m ()
killThread :: ... => ThreadId m -> 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 *masking state*, which can be used to block exceptions from other threads. There are three masking states: *unmasked*, in which a thread can have exceptions thrown to it; *interruptible*, in which a thread can only have exceptions thrown to it if it is blocked; and *uninterruptible*, in which a thread cannot have exceptions thrown to it. When a thread is started, it inherits the masking state of its parent, and the `forkWithUnmask` function forks a thread and passes it a function which can be used to execute a subcomputation in the unmasked state. We can also execute a subcomputation with a new masking state:

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

STM can also use exceptions, with its `throwSTM` and `catchSTM` functions. If an exception propagates uncaught to the top of a transaction, that transaction is aborted.

## 4. Testing using Déjà Fu

Testing with Déjà Fu consists in writing a small concurrent computation to test, and some predicates over the return value and traces produced. Predicates may be lazy: they need not examine the entire output before determining whether the test has passed or failed.

```
import Test.DejaFu

runTest :: Eq a
    => Predicate a -> (forall t. Conc t a)
    -> Result a
```

The abstract `Conc t` type is one of the instances of MonadConc for testing purposes (there's also `ConcIO t` for computations which do IO). The locally quantified type `t` prevents mutable state from leaking out of the computation, similarly to the ST monad.

```
type Predicate a
    = [(Either Failure a, Trace)] -> Result a

data Result a = Result
  { _pass         :: Bool
  , _casesChecked :: Int
  , _casesTotal   :: Int
  , _failures     :: [(Either Failure a, Trace)]
  } deriving (Show, Eq)
```

A `Result` consists of a Boolean flag indicating whether the test passed, the number of results checked to arrive at that conclusion, the total number of results, and a list of all failing cases.

Helper functions lift predicates over a single result to predicates over the collection:

```
alwaysTrue
    :: (Either Failure a -> Bool) -> Predicate a
somewhereTrue
    :: (Either Failure a -> Bool) -> Predicate a
```

There are also variants which take binary predicates for checking properties over the entire collection as a whole, for example:

```
alwaysSame :: Eq a => Predicate a
alwaysSame = alwaysTrue2 (==)

alwaysTrue2
    :: (Either Failure a -> Either Failure a -> Bool)
    -> Predicate a
somewhereTrue2
    :: (Either Failure a -> Either Failure a -> Bool)
    -> Predicate a
```

The functions `alwaysTrue2` and `somewhereTrue2` only check the predicate between values adjacent in the result list. The order of this list depends on the scheduling algorithm used, and so these functions should only be used for properties which are symmetric and transitive. There is also a collection of standard predicates, for doing things like checking for the existence of deadlock.

For the common case of checking for determinism, deadlock freedom, and proper exception handling, there is an `autocheck` function:

```
autocheck :: (Eq a, Show a)
    => (forall t. Conc t a) -> IO Bool
```

Let's see what we get from testing the example from the start of this paper. We'll have it return the value, rather than print it, though:

```
bad :: MonadConc m => m Int
bad = do
  shared <- newCVar 0
  fork . void $ swapCVar shared 1
  fork . void $ swapCVar shared 2
  readCVar shared
```

Firstly let's put it through `autocheck`:

```
> autocheck bad
[pass] Never Deadlocks (checked: 23)
[pass] No Exceptions (checked: 23)
[fail] Consistent Result (checked: 1)
  0 S0-------
  2 S0----P2---S0---
  1 S0----P1---S0---
False
```

Déjà Fu reports three distinct failures! Two failures are distinct if they have different results, or the same result but one trace is not a simplification of another. For each failure, the result is shown, along with the trace that led to it. "Sx" means that thread "x" started execution; "Px" means that thread "x" pre-empted the running thread; and the number of dashes indicates how many steps each thread ran for. So this output means that we get a "0" if there is no pre-emption, a "1" if thread 1 pre-empts the initial thread before the read, and a "2" if thread 2 pre-empts the initial thread before the read.

This output is nice for automated test suites, but perhaps not so friendly for interactive debugging. There are functions to run tests and return a more detailed result:

```
> runTest alwaysSame bad
Result {_pass = False, _casesChecked = 1
       , _casesTotal = 23, _failures = [...]}
```

The Result value tells us testing failed after looking at 1 case, there are 23 cases in total, and there is a simplified list of failures. These are quite long, so here is the second only:

```
( Right 2
, [ (Start 0,[],New 1)
  , (Continue,[],Put 1 [])
  , (Continue,[],Fork 1)
  , (Continue,[SwitchTo 1],Fork 2)
  , (SwitchTo 2,[Continue,SwitchTo 1],Take 1 [])
  , (Continue,[SwitchTo 0,SwitchTo 1],Put 1 [])
  , (Continue,[SwitchTo 0,SwitchTo 1],Stop)
  , (Start 0,[Start 1],Read 1)
  , (Continue,[SwitchTo 1],Lift)
  , (Continue,[SwitchTo 1],Stop)])
```

We have the result returned, and a log of what decision the scheduler made at each step, what alternative decisions it *could* have made, and what the thread did. Each thread and CVar (and CRef/CTVar) has its own unique identifier. In particular, we can see that when thread 2 pre-empted thread 0, it modified CVar 1, and there is evidence to suggest that the final result was determined by CVar 1 (as it was read just before the main thread terminated), this should suggest to us that maybe CVar 1 is the culprit, and lead us to look more closely at that area of the program.

It may seem difficult to keep track of which thread is which. Studies have found[16] that many errors are exposed with as few as two threads. This finding encourages us to write small test cases. However, there could be an optional mechanism to assign names to threads.

We can of course test STM transactions individually. The result may be a success (along with the value returned), a failure due to a `retry`, or a failure due to an uncaught exception:

```
import Test.DejaFu.STM
runTransaction :: (forall t. STMST t a) -> Result a
```

The abstract `STMST t` type is the testing implementation of MonadSTM.

### 4.1 Testing Aids

Some other functions generically defined for the typeclass only alter the running of the code during testing. They are provided to make it easier to write good tests.

Firstly, there is `_concNoTest`, used to indicate that a subcomputation already has its own tests, and so it should not be tested again *here*. Specifically, the test runner executes the argument of `_concNoTest` atomically. This allows tests to compose without repeating work:

```
big :: MonadConc m => m Int
big = do
  a <- _concNoTest little1
  b <- _concNoTest little2
  combine a b
```

If `little1` and `little2` already have their own tests, and have been verified to work, there is no need to test them again when testing `big`.

Secondly, there are cases where the main thread may be blocked on some CVar or CTVar, to which no other thread has a reference. This should cause immediate failure with deadlock as the reason. By default Déjà Fu cannot detect deadlocks of this kind. However, the user has the option to provide extra information, indicating which shared variables a thread knows about:

```
bad :: MonadConc m Int
bad = do
  _concAllKnown
  a <- newEmptyCVar
  b <- newEmptyCVar
  fork $ do
    _concKnowsAbout (Left a)
    _concAllKnown
    let loop = takeCVar a >> loop in loop
  fork $ do
    _concKnowsAbout (Left a)
    _concAllKnown
    let loop = putCVar a 1 >> loop in loop
  takeCVar b
```

The main thread is blocked on "b", to which neither of the other two threads has a reference. By adding annotations, this can be detected and reported. There are three annotations: `_concKnowsAbout` records that the current thread has a reference to a CVar or CTVar; `_concForgets` records that the current thread will never touch the referenced CVar or CTVar again; and `_concAllKnown` indicates that all CVars or CTVars which were passed in to the thread have been recorded. If every thread is in a known state, then detection of non-global deadlock is enabled. Otherwise the example above never terminates, as the two forked threads run forever, even though the main thread can never progress.

Misuse of these aids can lead to invalid test results. In particular, `_concNoTest` should only be used for actions which involve no shared variables from a larger scope. If two threads with a reference to the same shared variable are executed under `_concNoTest`, then the test runner will not consider possible interleavings of those threads.

### 4.2 IO

By itself, MonadConc cannot do IO. However, by adding in a MonadIO context and applying `liftIO` as appropriate, concurrency can be separated from other IO, allowing testing.

However, once IO is involved, the test runner loses control of what's going on. If a thread, during some IO, blocks on the action of another thread, this cannot be detected, and deadlock may arise. Furthermore, it is assumed for testing that the only source of nondeterminism is the scheduler (see §5). Any IO that is done should be deterministic given the same set of scheduling decisions, to not invalidate test results, although this is good practice in any sort of testing. Finally, the test runner cannot pre-empt within `liftIO` blocks, they should be as small as possible to avoid the risk of obscuring bugs.

## 5. Implementation

Readers who just want to use the Déjà Fu library can skip over this section and go straight to the example applications in §6.

### 5.1 Systematic Concurrency Testing

Systematic concurrency testing[16] (SCT) is a method for testing concurrent programs which works by forcing a particular set of scheduling decisions to be made. Different schedules can then be explored in order to try to find bugs. It is *systematic* because the order of exploration is generally not random, but follows some deterministic search pattern.

SCT can be implemented by overriding the concurrency primitives of the programming language, forcing all threads to run on one controlling thread, and making scheduling decisions between *effectively-atomic* blocks. An effectively-atomic block consists of a number of thread-local actions followed by a single access to a shared resource. This is preferable to exploring all possible points for making scheduling decisions, as the order of interleaving of thread-local operations cannot affect the final result.

There is some terminology associated with scheduling:

**Blocked** A thread awaiting access to a shared resource which is currently unavailable.

**Blocking** An operation which may result in its thread becoming blocked.

**Pre-emption** Pausing the execution of a thread which is not blocked, and executing another in its place.

**Pre-emption count** The number of pre-emptions in a schedule.

One common SCT algorithm is *pre-emption bounding*[13], where all schedules with a fixed pre-emption count are explored. A variant is *iterative pre-emption bounding*, where all schedules with a count of 0, and then 1, and so on up to the limit, are explored. A common bound chosen is 2, as empirical studies have found that many concurrency errors arise within that limit[13][16].

Another common algorithm is *delay bounding*[4], which explores schedules with a fixed number of deviations from an otherwise deterministic scheduler. This tends to perform about as well as pre-emption bounding in terms of finding bugs[16] although the number of schedules to explore grows slower, leading to more rapid testing. Despite this, it is not used in Déjà Fu as it is difficult to intuitively relate the lack of bugs found to some sort of correctness criteria for the program, unlike pre-emption bounding which has a direct relation to the complexity of the run-time behaviour.

### 5.2 Primitive Actions and Threading

The Conc and ConcIO monads represent threads as continuations over primitive actions, with the entire computation actually happening in a single Haskell thread. The primitives actions are shown, in abbreviated form, in Figure 1. Execution is more similar to co-operative multitasking than pre-emptive multitasking on a single processor. If executing a primitive action fails to terminate, the entire computation would lock up.

There are also a few other primitives omitted here for brevity, which are introduced by evaluating other primitives (for example, resetting the masking state). Execution happens in the context of an underlying monad, which implements mutable variables. For `Conc t` this is `ST t`, hence the type parameter. For `ConcIO t` it is `IO`, the parameter is retained to keep types similar.

Threads are stored in a map, from thread IDs to a record of the current state:

```
data Thread ... = Thread
  { _continuation :: Action ...
  , _blocking     :: Maybe BlockedOn
  , _handlers     :: [Handler ...]
  , _masking      :: MaskingState
  }
```

Evaluation is defined as repeating a single-step function until the main thread terminates, or deadlock is detected.

### 5.3 Shared State and Blocking

CRefs and CVars are both implemented in terms of the reference type of the underlying monad, as a pair (id, value), where CVars have a Maybe value.

```
data BlockedOn =
    OnCVarFull CVarId
  | OnCVarEmpty CVarId
  | OnCTVar [CTVarId]
  | OnMask ThreadId deriving Eq
```

```
data Action ... =
    AFork      thread_action action
  | AMyTId     (thread_id -> action)
  | APut       cvar new_value action
  | ATryPut    cvar new_value (Bool -> action)
  | AGet       cvar (value -> action)
  | ATake      cvar (value -> action)
  | ATryTake   cvar (Maybe value -> action)
  | AReadRef   cref (value -> action)
  | AModRef    cref function (result -> action)
  | AAtom      stm_action (result -> action)
  | ANew       action
  | ANewRef    action
  | ALift      (underlying_monad action)
  | AThrow     SomeException
  | AThrowTo   thread_id SomeException action
  | ACatching  handler action (result -> action)
  | AMasking   mask_state action (result -> action)
  | AStop

data STMAction ... =
    ACatch   action handler (result -> action)
  | ARead    ctvar (value -> action)
  | AWrite   ctvar new_value action
  | AOrElse  action action (result -> action)
  | ANew     action
  | ALift    (underlying_monad action)
  | AThrow   SomeException
  | ARetry
  | AStop
```

**Figure 1: Primitive actions**

When a CVar is accessed, the running thread is blocked if the CVar is in an inappropriate state. Otherwise the action of the thread is replaced with the relevant continuation. If the CVar has been mutated, then all threads blocked on reading that CVar (if it was put in to) or writing (if it was taken from) are unblocked. This unblocking behaviour is slightly different to MVars, where the order of awakening is FIFO.

The implementation of manipulating thread block statuses is shown in Figure 2. Note the special case in `wake` for being blocked on a collection of CTVars: if there is *any* intersection between the lists of CTVars, the thread is woken.

### 5.4 Exceptions

A thread has a stack of exception handlers. Upon entering a `catch`, the handler is pushed to the stack, and a primitive action to pop it is inserted at the end of the enclosed action. A handler, when invoked, replaces the action of the thread entirely, jumping to the continuation of the `catch` after the programmer-supplied function terminates:

```
data Handler ... = forall e. Exception e
    => Handler (e -> Action ...)
```

Upon evaluating a `throw`, the exception handler stack is popped until a handler capable of handling the exception is reached. The action of the thread is then replaced with the handler, and the new stack is stored. If no handler is found, the thread is killed. If this is the main thread, the entire computation terminates with an error.

When a mask is entered, a primitive action to restore the masking state is added on to the end of the subcomputation.

```
block :: BlockedOn -> ThreadId -> Threads n r s
    -> Threads n r s
block blockedOn = M.alter doBlock
  where
  doBlock (Just thread) = Just $
    thread { _blocking = Just blockedOn }

wake :: BlockedOn -> Threads n r s
    -> (Threads n r s, [ThreadId])
wake blockedOn threads =
  ( M.map unblock threads
  , M.keys $ M.filter isBlocked threads )

  where
  unblock t
    | isBlocked t = t { _blocking = Nothing }
    | otherwise   = t

  isBlocked t = case (_blocking t, blockedOn) of
    (Just (OnCTVar ctvids), OnCTVar blockedOn')
      -> ctvids `intersect` blockedOn' /= []
    (theblock, _) -> theblock == Just blockedOn
```

**Figure 2: Manipulating thread blocks**

### 5.5 Software Transactional Memory

STM is implemented in terms of its own primitive actions, also shown in Figure 1. CTVars are implemented in terms of STRefs, if using Conc, or IORefs, if using ConcIO.

As STM transactions are atomic, the implementation is quite simple. It repeats a single-step function until the transaction reaches a fixed point: an `ARetry`, `AThrow`, or `AStop` action. Only the `AStop` action indicates successful termination. If a transaction terminates due to an `ARetry` action, the thread is blocked.

Executing an STM transaction returns a result (or indication of failure), a list of CTVars written to (if success) or read from (if failure), and an action in the underlying monad to undo the effects of the transaction.

An `ACatch` action is implemented by simply executing the entire subcomputation and pattern matching on the return value: if it is a success, the value is returned; if it is an exception of the appropriate type, it is passed to the handler; and if it is a different exception, it is propagated upwards.

### 5.6 Detecting Deadlock

Deadlock detection is implemented in GHC as part of garbage collection: if a thread is blocked on a variable to which no running thread has a reference, that thread is deadlocked. Unfortunately, the garbage collector is beyond the reach of Déjà Fu (and even if it wasn't, would require everything to be in IO). So by default in Déjà Fu the only deadlock detection is global: where every thread is blocked simultaneously.

Deadlock where the main thread is blocked on a shared variable for which no other thread has a reference is optionally implemented with special _conc functions. See §4. These record for each thread which shared variables are known about, allowing largely the same approach as the GC one if the state of every thread is fully known. However if these functions are incorrectly used, there may be false results of testing.

### 5.7 Schedule Bounding

Testing in Déjà Fu is, by default, implemented using pre-emption bounding with a bound of two. Other bounds can be set. Also, enough of the internals are exposed such that other SCT runners could be implemented.

An execution is parameterised with a deterministic scheduler which may have some state. The execution returns the result, an execution trace, and the final scheduler state. Using the scheduler state, we can implement a very simple scheduler which takes some list of initial decisions to make (a schedule prefix), and which makes non-pre-emptive decisions after that point.

Schedule bounding generates new schedules from a schedule prefix and suffix. Given a schedule suffix, there are functions to generate *siblings* and *offspring*. A sibling is a new partial prefix which, when appended to any prefix at all, does not result in a prefix in a different bounding level. An offspring is a new partial prefix which, when appended to any prefix at all, results in a prefix in the next bounding level up. In the case of pre-emption bounding, siblings are partial prefixes with no pre-emptions, and offspring are partial prefixes with one pre-emption, so producing a prefix with $n+1$ pre-emptions when appended to the original prefix.

***Example***

```
prefix = [Start 0]
suffix = [(Continue, [], Fork 1)
         ,(Continue, [SwitchTo 1], Stop)]
```

Given this prefix and suffix, under pre-emption bounding there are no siblings, as the only available alternative choice would introduce a pre-emption. There is one offspring, by making the alternative decision at step 2 of the suffix:

```
siblings suffix == []
offspring suffix == [[Continue, SwitchTo 1]]
```

This offspring would not actually be generated, however. Pre-emptions are only introduced around actions such as access to a CVar, where pre-emption may affect the final result. □

This splitting into prefixes and suffixes makes it easy to prevent duplicate schedules. The schedule bounding runner stops generating offspring when the bound is reached, and explores schedules in a mostly breadth-first fashion. Furthermore, there is an option to explore all schedules.

Also implemented is a delay-bounding scheduler. A *delay* is a deviation from an otherwise deterministic scheduler. So delay-bounding has the advantage that the number of schedules grows more slowly than pre-emption bounding: there is exactly one schedule with a delay count of 0, but potentially many with a pre-emption count of 0. The default testing mechanisms use pre-emption bounding because the guarantees that delay-bounding gives are influenced by the choice of scheduler, whereas pre-emption bounding gives a global property of all schedules. The two methods tend to perform about the same in terms of bug-finding ability[16].

## 6. Examples

Four examples are discussed, two of which are external libraries. The first is a variation of an example in *Parallel and Concurrent Programming in Haskell*[10] of a concurrent message logger, into which a bug has intentionally been introduced. The entire program is presented, as it is small. Then two known bugs in the *auto-update* package are reproduced, and one of the schedulers in the *monad-par* package is tested. The last is a bug that arose, unintentionally, in the implementation of a library for performing search problems in parallel, where an incorrect use of CTMVars allowed a user of the library to obtain an incomplete result.

### 6.1 Message Logger

Suppose we want a concurrent message logger with the following properties:

- The logger can be sent a message, or it can be told to stop; when told to stop, all messages sent before that point are returned to the thread which stopped it.
- Messages from the same thread should be in order, but messages from different threads may be in any order.

Firstly, we shall define the types we're going to use:

```
data Logger m    = Logger (CVar m LogCommand)
                          (CVar m [String])
data LogCommand = Message String | Stop

initLogger :: MonadConc m => m (Logger m)
initLogger = do
  cmd <- newEmptyCVar
  log <- newCVar []
  let l = Logger cmd log
  fork $ logger l
  return l
```

Now we need to be able to send a message to the logger. As CVars are being used, these functions will block if there is already a command there. We need not worry about threads overwriting each other's commands.

```
logMsg :: MonadConc m => Logger m -> String -> m ()
logMsg (Logger cmd _) = putCVar cmd . Message

logStop :: MonadConc m => Logger m -> m [String]
logStop (Logger cmd log) = do
  putCVar cmd Stop
  readCVar log
```

Finally, we have the main loop of the logger. It blocks on taking a command. If the communication is a new message, the logger appends the message to the list and loops, otherwise it terminates.

```
logger :: MonadConc m => Logger m -> m ()
logger (Logger cmd log) = loop where
  loop = do
    command <- takeCVar cmd
    case command of
      Message str -> do
        strs <- takeCVar log
        putCVar log $ strs ++ [str]
        loop
      Stop -> return ()
```

If at least two threads attempt to communicate with the logger after it has been stopped, one will block indefinitely. We assume one supervising process which orchestrates the concurrency (for example, a managing thread which starts a logger and a collection of worker threads, which report their status to the log), so this isn't a problem.

The actual bug is less obvious, so let's write a simple test case and see what `autocheck` does for us:

```
test :: MonadConc m => m[(ThreadId m, String)]
test = do
  l <- initLogger
  j1 <- spawn (logMsg l "a" >> logMsg l "b")
  j2 <- spawn (logMsg l "c" >> logMsg l "d")
  readCVar j1; readCVar j2
  logStop l
```

Here we start a logger, fork off two threads which each write two messages to the log, wait for them to terminate, and stop the logger. We should always see 4 log entries, with "a" before "b", "c" before "d", but all other orderings.

Running with `autocheck`, we see[1] the following:

```
> autocheck test
[pass] Never Deadlocks (checked: 104626)
[pass] No Exceptions (checked: 104626)
[fail] Consistent Result (checked: 5)
  ["a","b","c"] S0---------S2-----S3---S1----S2--
    -S1----S3---S1----S3---S1-P0------
  ["a","b","c","d"] S0---------S2-----S1----S2---
    S1----S3-----S1----S3---S1----S0------
  ["a","b","c"] S0---------S2-----S1----S2---S1--
    --S3-----S1----S3---S0---S1-P0----
  ["a",c","b"] S0---------S2-----S1----S3-----S2-
    S1----S2---S1----S3---S1-P0------
  ["a","c","b","d"] S0---------S2-----S1----S3---
    --S1----S2---S1----S3---S1----S0------
  ...
False
```

Well, we found a bug: sometimes the last message gets missed. Also, the cases where the last message is dropped all appear to end with a pre-emption of thread 1 by thread 0. As threads are numbered sequentially in order of creation, thread 0 is the initial thread and thread 1 is the logger thread. We can restrict the results by checking a different condition:

```
> dejafu test
  ( "4 Values"
  , alwaysTrue $ \(Right xs) -> length xs == 4)
[fail] 4 Values (checked: 16)
  ["a","b","c"] S0---------S2-----S3---S1----S2--
    -S1----S3---S1----S3---S1-P0------
  ["a","b","c"] S0---------S2-----S1----S2---S1--
    --S3-----S1----S3---S0---S1-P0----
  ["a","c","b"] S0---------S2-----S1----S3-----S2
    -S1----S2---S1----S3---S1-P0------
  ["a","c","d"] S0---------S2-----S1----S3-----S1
    ----S3---S2-S1----S2---S1-P0------
  ["a","c","b"] S0---------S2-----S1----S3-----S1
    ----S2---S1----S3---S0---S1-P0----
  ...
False
```

The pattern continues. Upon a closer inspection of `logger`, we can see that if it is pre-empted between the `takeCVar cmd` and the `takeCVar log`, a stop command can be written without blocking, and an incomplete log returned. One solution would be to replace the first `takeCVar` with a `readCVar`, and only empty the CVar when the processing of the command is complete.

### 6.2 The auto-update Package

The *auto-update*[2] 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[15]. Despite the core functionality being very simple, two race conditions were noticed by users inspecting the code in October 2014.

[1] Traces have been broken into multiple lines here, but the tool does not do any output wrapping by itself.

[2] `https://hackage.haskell.org/package/auto-update`

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

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

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

    void $ forkIO $ forever $ do
        takeMVar needsRunning

        a <- catchSome $ updateAction us

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

        threadDelay $ updateFreq us

        writeIORef currRef Nothing
        void $ takeMVar lastValue

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

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

**Figure 3:** ***auto-update*** **implementation**

The entire implementation, excluding comments and imports, is reproduced in Figure 3. 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 => m ()
test = do
  auto <- mkAutoUpdate defaultUpdateSettings
  auto
```

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

```
> autocheck test
[fail] Never Deadlocks (checked: 1)
        [deadlock] S0-------S1---------S0-
[pass] No Exceptions (checked: 17)
[fail] Consistent Result (checked: 4)
        () S0-------S1------P0---
        [deadlock] S0------P1---------S0-
        () S0-------S1-------P0---
False
```

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 => 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:

```
> dejafus' 3 test [("Consistent Result", alwaysSame)]
[fail] Consistent Result (checked: 5)
        0 S0--------S1-------P0-----
        [deadlock] S0-------P1----------S0-
        1 S0--------S1-------P0--P1---S0---S1----
          ---P0----
        1 S0--------S1-------P0--P1---S0---S1----
          ----P0----
        0 S0--------S1--------P0--------
        ...
False
```

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!

### 6.3 Parallel Search

The *Search Party*[1] 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:

[1] `https://github.com/barrucadu/search-party`

```
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:

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

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

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

[fail] Result Lists (checked: 46)
  Just [2,1] S0----S1-----S2-P3-----------S0----
  Just [0,2,1] S0----S1-----S2-P3-----------S2---
    ---S0----
  Just [2,1] S0----S1-----S2--P3-----------S0----
  Just [0,2,1] S0----S1-----S2--P3-----------S2--
    ---S0----
  Just [2,1] S0----S1-----S2---P3-----------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.

### 6.4 The Par Monad

As mentioned in Section §2.2, the Par monad allows for deterministic data-flow parallelism in Haskell. 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[1], 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*[1] package and two from the *monad-par*[2] package.

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

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

Some simplifications were made in the conversion process:

- Par normally uses the *mwc-random*[3] package when performing its internal scheduling. This was initially replaced with a constant function, and then a `StdGen`.
- Behaviour of the Par scheduler is configured by cpp, but only the default configuration was tested.

Figure 4 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:

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

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

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.

## 7. Related Work

Pre-emption bounding testing tools exist for both C[17] and Java[4] at least. SCT in Java is particularly nice, as the bytecode can be instrumented to support SCT at runtime, by the test runner. This frees the programmer from the need to structure their code in such a way to support SCT, they can just program as they have always done. This also allows legacy concurrent applications to be tested easily.

PULSE[3] is a concurrency testing tool for Erlang, where processes are synchronised by communicating with a scheduler process, and QuickCheck is used for schedule generation. PULSE supports automatic code instrumentation to enable this style of testing. As Erlang processes may be distributed, pre-emption bounding may not be suitable, as it assumes everything is executing on a single processor. There has been work on Erlang-style concurrency for Haskell[5]. It seems to be little used, but in this case a PULSE-style approach may be better.

[1] `https://hackage.haskell.org/package/abstract-deque`

[2] `https://hackage.haskell.org/package/monad-par`

[3] `https://hackage.haskell.org/package/mwc-random`

[4] LazyLocks (Paul Thomson), to appear.

```haskell
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,
                 rngs=rng, sessions=stck,
                 activeSessions=activeSessions,
                 sessionCounter=sessionCounter
               }
         | x    <- [0 .. caps-1]
         | wp   <- workpools
         | rng  <- rngs
         | stck <- sessionStacks
       ]
  return allscheds
```

**Original**

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

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

**Déjà Fu**

**Figure 4: Par "direct" scheduler initialisation**

Whilst the MonadConc typeclass was structured to be similar to the standard concurrency primitives, the inspiration for this approach, and the basic idea behind how to do SCT in Haskell, was provided by a blog post[2]. However, both the family of primitives and the approach to testing have been significantly advanced.

## 8. Conclusions & Further Work

Although a commonly reported experience amongst Haskell programmers is that "if it compiles, it works", there are times where it does *not* work. A number of profiling and debugging tools exist, typically requiring special runtime support. Concurrency is a particularly difficult area to get right, as everyone who has had to move outside the realm of guaranteed determinism will know. Yet there are no debugging tools for concurrent Haskell programs (ThreadScope[8] is a profiling tool, and merely gathers information on sample executions). This paper contributes such a tool, at the cost of a programmer having to use a generalisation of the familiar concurrency abstraction.

Is this cost too high? Programmers are notoriously unwilling to restructure their code to allow for easier analysis or testing, unless the current situation is truly unbearable. It is generally regarded in the Haskell community as good practice to write IO-using functions as thin wrappers around calls to pure code. This practice should limit the amount of change needed. The MonadConc and MonadSTM interfaces have been kept intentionally very similar to the IO and STM interfaces. Typically all that a programmer needs to do is to change some imports, some names, and a few type signatures.

It is impossible in the current implementation to include functions like `threadDelay`, as testing assumes that any nondeterminism is due to the scheduler. Causing a thread to sleep is a notoriously nondeterministic operation, as the actual amount of time slept depends partly on the operating system's scheduler, which remains out of reach.

If a thread enters an infinite loop in a primitive action call, the entire test runner will lock up, even if that would not happen when executing normally. This is because the test runner cannot do things on a granularity smaller than one primitive action.

Despite these limitations, our tool solves a problem, and makes writing reliable Haskell programs a little easier.

We implemented a library for fast parallel search on top of this abstraction, and some shortcomings were identified and rectified. In particular, there was originally no CRef type, as IORefs operations can potentially be re-ordered[7]. However, IORefs with non-reorderable updates turned out to be exactly the abstraction needed for *Search Party*'s work stealing scheduler, and so they were added.

More could be done. For example:

- Swapping out the regular concurrency primitives for the MonadConc primitives could be done at compile- or link-time, as a GHC plugin, rather than at the level of code. This would allow testing of legacy code, and also free the programmer from needing to modify their code. However, it would require recompiling all dependencies with this functionality enabled.
- Dynamic partial-order reduction (DPOR)[6] is a technique for dynamically deciding which traces will not be interesting based on thread interactions, and so greatly reducing the search space. This would increase testing performance, and make feasible the testing of large programs.
- In practice, schedulers are biased towards a particular subset of the possible schedules. They may try to guarantee fairness, for example. At the cost of less complete results, schedules which are not sufficiently fair could be ignored, reducing the search space.

## References


[1] Parallel and Concurrent Programming in Haskell (online version, part of Atlas beta), 2014. URL `https://www.reddit.com/r/haskell/comments/1iwr7x/parallel_and_concurrent_programming_in_haskell/cb8x76p`.

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

[3] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 149–160. ACM, 2009.

[4] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded Scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 411–422. ACM, 2011.

[5] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129. ACM, 2011.

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

[7] GHC Base Libraries. Data.IORef, 2015. URL `https://hackage.haskell.org/package/base-4.7.0.2/docs/Data-IORef.html#g:2`.

[8] D. Jones Jr, S. Marlow, and S. Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 81–92. ACM, 2009.

[9] L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 2–14. ACM, 2014.

[10] S. Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. O'Reilly Media, 2013. ISBN 9781449335922.

[11] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq No More: Better Strategies for Parallel Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 91–102. ACM, 2010.

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

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

[14] E. Scholz. *A Concurrency Monad Based on Constructor Primitives, or, Being First-Class is Not Enough*. Freie Univ., Fachbereich Mathematik, 1995.

[15] M. Snoyman. Announcing auto-update, 2014. URL `http://www.yesodweb.com/blog/2014/08/announcing-auto-update`.

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

[17] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 485–502. ACM, 2012.