Revealing Behaviours of Concurrent Functional Programs by Systematic Testing

Michael Stewart Walker

Doctor of Philosophy

University of York
Computer Science

March 2018

Abstract

We aim to make it easier for programmers to write correct concurrent programs and to demonstrate that concurrency testing techniques, typically described in the context of simple core languages, can be successfully applied to languages with more complex concurrency. In pursuit of these goals, we develop three lines of work:

Testing concurrent Haskell We develop a library for testing concurrent Haskell programs using a typeclass abstraction of concurrency, which we give a formal semantics. Our tool implements *systematic concurrency testing*, a family of techniques for deterministically testing concurrent programs. Along the way we also tackle how to soundly handle daemon threads, and how to usefully present complex execution traces to a user. We not only obtain a useful tool for Haskell programs, but we also show that these techniques work well in languages with rich concurrency abstractions.

Randomised concurrency testing We propose a new algorithm for *randomly* testing concurrent programs. This approach is fundamentally incomplete, but can be suitable in cases where systematic concurrency testing is not. We show that our algorithm performs as well as a pre-existing popular algorithm for a standard set of benchmarks. This pre-existing algorithm requires the use of program-specific parameters, but our algorithm does not. We argue that this makes use and implementation of our algorithm simpler.

Finding properties of programs We develop a tool for finding properties of sets of concurrency functions operating on some shared state, such as the API for a concurrent data type. Our tool enumerates Haskell expressions and discovers properties by comparing execution results for a variety of inputs. Unlike other property discovery tools, we support side effects. We do so by building on our tool for testing concurrent Haskell programs. We argue that this approach can lead to greater understanding of concurrency functions.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

For my family; my mum Jill and brother Mason but especially my dad Mark, without whose support I would not have got this far.

About a year and a half into my Ph.D, I decided that I had had enough and wanted to quit. I was feeling burned out over what felt like doing the same thing over and over again, and over a conflict of motivation I had come to recognise between myself and academia. I am motivated by making and maintaining tools which people use, whereas academia is motivated by finding novel results, and the two do not align perfectly. Fortunately, with the encouragement of Colin, my supervisor, I decided to give it until after the summer to set any wheels in motion. That break was what I needed, and I came back able to stick it through to the end.

I did not receive any funding to do my Ph.D. Due to the generous support of my family, and the six months I took off for internships, I was able to work on my Ph.D full time despite the financial handicap. Having to self-fund a degree is a bit of a shock which forces you to become good with money. Even though it was hard at times—because it was hard at times—I think this is one of the more directly valuable skills I gained during my time. I only wish I'd learned this as an undergraduate, when I really didn't need to spend as much as I did.

The PLASMA group has been a constant source of fun and of knowledge, even though I got into the habit of going into the office at strange hours, often not overlapping much with anyone else. Thanks to José for guiding me through the strange ways of being a Ph.D student in my first year, and to Rudy and Matt for letting me bounce ideas off them. It's a shame that with a few of us leaving in quick succession, PLASMA is now so small.

Thanks to my friends both in-person and online, for putting up with my venting on more than one occasion. Finally, thanks to those who remind me that, no matter what is going on at the moment, one should always strive to take it easy.

Declaration

This work has not previously been presented for an award at this, or any other, university. All sources are cited in the main text and listed in the bibliography. Earlier versions of parts of this thesis were published in the following papers:

1. Michael Walker and Colin Runciman. Déjà Fu: A Concurrency Testing Library for Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 141–152. ACM, 2015.
2. Michael Walker and Colin Runciman. Cheap Remarks about Concurrent Programs. Presented at *Trends in Functional Programming*. 2017.
3. Michael Walker and Colin Runciman. Cheap Remarks about Concurrent Programs. In *Functional and Logic Programming Symposium*, FLOPS 2018, pages 264–279. Springer, 2018.

These papers were conceived, implemented, and written by myself with significant input from Prof. Runciman. The first paper contributes to Chapter 5, the second and third to Chapter 7. An updated version of the first paper was published as a departmental technical report:

4. Michael Walker. Déjà Fu: A Concurrency Testing Library for Haskell. Technical report, University of York, Department of Computer Science, 2016.

The inspiration for the investigation in Chapter 6 arose from discussions with Dr. Alistair Donaldson and the now-Dr. Paul Thomson[1].

[1] Both at Imperial College London

Chapter 1

Introduction

There is a tension between theory and practice in software engineering. Whenever one programmer suggests some sophisticated technique or formalism, another will question its applicability to the *real world*, a nebulous and under-specified place. For example, a common argument against techniques such as model checking is that they do not scale to real-world problems, whatever those are, and that testing is good enough. Testing, aided by dynamic analyses like Clang's ThreadSanitizer (Serebryany and Iskhodzhanov 2009) or Go's data race detector (The Go Developers 2017a), is often effective at finding flaws, and is familiar to most programmers. However, unlike testing, model checking can prove the absence of bugs, though it can be difficult or resource-intensive to use.

In this thesis, we are interested in the testing of concurrent programs. In this setting, testing falls down when considering questions like:

- How do we ensure that we're covering a variety of schedules?
- How do we know a that bug has been fixed?
- How much testing is enough?

We try to follow a middle path between the familiarity of testing and the power of model checking. By using *systematic concurrency testing*, we enable programmers to test their concurrent programs deterministically, and confidently.

A concern in academia is the tension between theoretical novelty and practical utility. Too often programs written during research are abandoned as unpolished prototypes. This practice harms the spread of ideas from theory into practice, which is particularly regrettable when the ideas are intended to make programmers' jobs easier. We adopt a stance more in favour of practical utility than is perhaps typical. By producing polished and featureful tools, we enable programmers to benefit immediately from our work.

We use Haskell as the implementation language, and the source of the concurrency abstraction we target, in this thesis. However, our work is not tied to Haskell. Appendix A gives a brief introduction to Haskell for the reader already somewhat familiar with functional programming in other languages.

1.1. Parallelism vs Concurrency

The terms *parallelism* and *concurrency* are broadly, but not quite, synonymous. Following the lead of Peyton Jones, Gordon, and Finne 1996, we use them to refer to different but related concepts:

Parallelism A parallel program uses a multiplicity of hardware to compute different aspects of a result simultaneously. The goal is to arrive at the overall result more quickly. For example, the x86 assembly instruction `PMULHUW` computes the element-wise multiplication of two vectors, performing each multiplication simultaneously: it enables parallelism.

Concurrency A concurrent program uses multiple threads of control to structure the program. These threads conceptually execute independently and at the same time. But whether threads do execute simultaneously is an implementation detail. A concurrent program can execute on a single-core machine through interleaved sequential execution just as it can execute on a multi-core machine in parallel. A concurrency abstraction can guarantee parallelism (given suitable hardware), for example by having the ability to restrict the execution of individual threads to given processor cores.

It is tempting to think of parallelism as being *semantically invisible*: not changing the result of a program, merely making it faster. However, on modern processors, parallelism is semantically *visible*. This thesis is mostly concerned with concurrency, but the *relaxed memory* behaviour of modern processors, an artefact of parallelism, appears in Chapter 5.

1.2. Goals and Contributions of this Thesis

The overall motivation of this research has been to develop tools which make it easier for programmers to write correct concurrent programs. Our primary goal is to demonstrate that concurrency testing techniques, typically described in the context of a simple core language, can be successfully applied to languages with rich concurrency abstractions.

Specifically, we want a concurrency testing tool which simultaneously supports multiple aspects of concurrency which are traditionally considered difficult: such as relaxed memory, software transactional memory, and inter-thread signals. Furthermore, we want to show that supporting these features does not render a tool too expensive, in time or space, to be of practical value.

To meet and demonstrate our objectives, we develop libraries and tools for testing Concurrent Haskell programs, and evaluate their effectiveness on sample applications.

Our major contributions are:

- A library for effectively testing Concurrent Haskell programs, in Chapter 5. We demonstrate its effectiveness with case studies of three concurrency-using Haskell libraries.
  There are no sound and complete concurrency testing tools for Haskell. Our tool fills this niche.
- A new scheduling algorithm for randomised testing to allow testing programs where complete testing does not scale, in Chapter 6. We evaluate its bug-finding ability on a standard set of benchmarks.
  Randomised testing can be both fast and effective, but existing algorithms such as PCT (Burckhardt et al. 2010) require difficult-to-obtain information about the program under test. Ours does not.
- A tool for discovering properties of Haskell functions operating on shared mutable state in the presence of concurrent interference, in Chapter 7. We give case studies of three concurrent data structures.
  Tools such as QuickSpec (Smallbone et al. 2017) and Speculate (Braquehais and Runciman 2017b) can discover properties of pure functions. Our approach extends this to nondeterministic and concurrent program fragments.

In achieving these major contributions, we also achieve smaller ones along the way:

- An abstraction over the GHC Haskell concurrency API, in Section 5.2.
  We use a typeclass abstraction permitting multiple possible implementations.
- An operational semantics for Concurrent Haskell, in Section 5.5.
  Our semantics is similar in spirit to Vollmer et al. 2017, however we model a much larger set of operations, and support more memory models.
- A method for soundly incorporating daemon threads into a concurrency testing setting, in Section 5.6.

Daemon threads are threads which automatically terminate when the main thread terminates. They are typically omitted from concurrency testing algorithms, and a straightforward implementation is too inefficient to be practical.

- An algorithm for semantics-preserving simplification of execution traces, in Section 5.7.

  These simplified traces are easier for a programmer to follow than a trace which may contain unnecessary details and scheduling decisions.
- Case studies of applying concurrency testing to three Haskell libraries, in Section 5.9.

  Through these case studies we demonstrate that our tools are of practical use to programmers.
- A method for generating program fragments containing lambda terms, in a restricted setting, in Section 7.3.1.

  Lambda terms are essential in generating complex monadic expressions.

1.3. Roadmap

Part I We present the context and background of the work. Chapter 2 gives an introduction to concurrency in Haskell. Chapter 3 discusses the theory behind *testing* concurrent programs. Finally, Chapter 4 gives an introduction to property testing in Haskell.

Part II We present our contributions. Chapter 5 gives an account of the Déjà Fu tool for testing concurrent Haskell programs. Chapter 6 discusses an alternative scheduling algorithm for testing concurrent programs. Finally, Chapter 7 gives an account of the CoCo tool for discovering properties of concurrent Haskell programs, and shows how it connects to Déjà Fu. These chapters build on each other:

- Déjà Fu, in Chapter 5, allows exploring the behaviours of a concurrent program, and on top of this foundation we build an equivalence checker.
- Swarm scheduling, in Chapter 6, is an alternative scheduling algorithm which can be plugged into Déjà Fu.
- CoCo, in Chapter 7, builds on the equivalence checker in Déjà Fu, by generating and testing candidate equivalences from programmer-supplied primitives.

Part III We present our overall conclusions in Chapter 8 and suggest possible future work in Chapter 9.

1.4. Source Code

The Déjà Fu and CoCo tools we develp in Part II are available on GitHub:

- `https://github.com/barrucadu/dejafu`
- `https://github.com/barrucadu/coco`

Déjà Fu and its related libraries are also available on Hackage:

- `https://hackage.haskell.org/package/concurrency`
- `https://hackage.haskell.org/package/dejafu`
- `https://hackage.haskell.org/package/hunit-dejafu`
- `https://hackage.haskell.org/package/tasty-dejafu`

Part I

Background and Review

Chapter 2

Concurrent Haskell

In this chapter we discuss the concurrency model of interest (§2.1), and then give an overview of Concurrent Haskell (Peyton Jones 2002; Peyton Jones, Gordon, and Finne 1996). We use Haskell as the implementation language in this thesis. Concurrency is not in the Haskell standard, the operations we discuss are available in GHC and may not be in other compilers. We cover the basic use of concurrency (§2.2), the memory model (§2.3), software transactional memory (§2.4), and exceptions (§2.5). We then walk through the development of a small example program (§2.6).

2.1. The Concurrency Model

Threads, locks, and shared variables are the bread and butter of concurrent programming. We are interested in models of concurrency more complex than this. In this thesis we focus on the model of Concurrent Haskell, which introduces a number of additional features:

- *Relaxed memory*, where operations on shared variables may appear to happen out-of-order.
- *Software transactional memory*, where groups of shared variables can be operated on together atomically.
- *Synchronous exceptions*, where an error in a thread can interrupt its flow of control.
- *Asynchronous exceptions*, where one thread can asynchronously interrupt the flow of control of another thread.

Concurrency testing algorithms are typically presented in the setting of a simple core concurrent language of just reads and writes. We choose Concurrent Haskell, and the

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

Listing 1: Basic threading operations in Haskell.

```java
/* forkIO */
Runnable runnable = /* action */;
Thread thread = new Thread(runnable);
thread.start();

/* myThreadId */
Thread me = Thread.currentThread();
```

Listing 2: Basic threading operations in Java.

features it provides, because it cannot be decomposed into this simple case.

These features are not unique to Haskell. Throughout, we compare with the concurrency abstractions of Java (Lea 1996) and Rust (The Rust Developers 2011). The overlap of functionality is not perfect—in some cases Haskell is more expressive, in others these languages are more expressive—but we hope to show that, although we use Haskell, there is nothing Haskell-specific in our results or methods.

2.2. Multithreading

Threads let a program do multiple things at once. Every program has at least one thread, which runs the main action of the program. A thread is the basic unit of concurrency.

Haskell's basic threading functions are shown in Listing 1. A thread can be started using the `forkIO` function, which starts executing its argument in a separate thread and also gives us back a `ThreadId` value, which can be used to kill the thread. A thread can get its own `ThreadId` using `myThreadId`.

In Java (Lea 1996), threads are created from classes implementing the `Runnable` interface, as shown in Listing 2. The `Thread` constructor creates a new thread object from a `Runnable`, but it does not start until `Thread.start` is called. The thread object itself fulfils the role of the Haskell `ThreadId` type. A thread can get a reference to itself with the `Thread.currentThread` static method.

Rust (The Rust Developers 2011) supports both the Haskell and Java thread creation styles, as shown in Listing 3. The Haskell-style `thread::spawn` function takes a closure to execute, creates and immediately begins executing a thread, and returns an identifier. The alternative Java-style `thread::Builder` interface allows creating a thread

```
/* Haskell style */
let thread = thread::spawn(/* closure */);

/* Java style */
let thread = thread::Builder::new().spawn(/* closure */);

/* myThreadId */
let me = thread::current();
```

Listing 3: Basic threading operations in Rust.

```
forkOn             :: Int -> IO () -> IO ThreadId

getNumCapabilities :: IO Int
setNumCapabilities :: Int -> IO ()
```

Listing 4: Operating system threads in Haskell.

without starting it. Rust enforces an *ownership type* system. The compiler gives an error if a thread closure captures a variable from its outer scope which is used later in the outer scope. Preventing variables from being used across scopes is the source of much of Rust's memory safety.

Capabilities In a real machine, there are multiple processors and cores. It may be that a particular application of concurrency is only a net gain if each thread is operating on a separate core, so that threads are not interrupting each other. GHC uses a *green threading* model, where Haskell threads are multiplexed onto a much smaller number of operating system threads (Marlow, Peyton Jones, and Singh 2009). The number of operating system threads is referred to as the number of *capabilities* or *Haskell execution contexts* (HECs) (Marlow, Peyton Jones, and Singh 2009). Only operating system threads have the possibility of executing truly in parallel.

We can fork a thread to run on a particular capability with the `forkOn` function, which takes a number identifying the capability to use. This capability number is interpreted modulo the total number of capabilities, which can be queried and set. Listing 4 shows the capability functions.

Neither Java nor Rust provide green threading. Java does not specify how its threads are mapped to OS threads but, on Linux, each Java thread is an OS thread. Rust specifies that its threads are OS threads.

```haskell
yield       :: IO ()
threadDelay :: Int -> IO ()
```

Listing 5: Controlling thread scheduling in Haskell.

```java
Thread thread = /* ... */;
thread.setPriority(/* new priority */);
```

Listing 6: Thread priority in Java.

Scheduling The GHC scheduler is necessarily general-purpose. However, sometimes we have domain knowledge which lets us do better.

Listing 5 shows the two ways to influence how threads are scheduled: (1) we can yield control to another thread, or (2) we can delay the current thread for a period of time.

In Java, we can use the `Thread.yield` and `Thread.sleep` methods, shown in Listing 6, to affect scheduling. We can also adjust the *priority* of a thread, where the initial priority is inherited from its creator. Threads with higher priority are executed in preference to threads with lower priority. Haskell threads have no notion of priority. However, GHC uses a round-robin scheduler, so no one thread can starve another.

```rust
thread::park() /* execution stops now */

/* from another thread */
reference_to_thread::unpark();
```

Listing 7: Thread parking and unparking in Rust.

Rust has three ways to control scheduling. In addition to yielding and delaying, it can also *park* the current thread, shown in Listing 7. When parked, a thread will not execute until it is unparked by another thread. There is a variant of `thread::park` with a timeout, which provides a delay-unless-woken construct.

Haskell threads have no notion of parking. However, parking is not an essential primitive. It can be implemented by associating an `MVar` (§2.3) with each thread. Parking corresponds to `takeMVar`. Unparking corresponds to `tryPutMVar`. Parking with a timeout can be implemented by forking a thread to unpark after a delay.

Termination Both Java and Rust can use a thread handle to block until that thread terminates. This is called *joining*. Haskell provides no join operation, but one can be implemented using by associating an `MVar` with each thread, like parking. Before a thread terminates it will execute a `putMVar`, and joining corresponds to `readMVar`.

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Listing 8: Shared mutable references in Haskell.

```
let ptr = &mut /* initial value */;
let shared = Arc::new(AtomicPtr::new(ptr));

let shared_clone = shared.clone();
let thread = thread::spawn(move || {
  shared_clone.store(/* new value */, Ordering::SeqCst);
});
```

Listing 9: Shared mutable references in Rust.

2.3. Shared State and the Memory Model

Concurrent Haskell uses a shared-memory model for communication between threads. There are two main types of shared variable, with different semantics.

Shared mutable references An `IORef` is a mutable location in memory holding a Haskell value. The API is shown in Listing 8.

Java is an impure language with no restriction on sharing, so it has no need for a type like `IORef`. Any thread can mutate any reference that is in scope.

Rust does impose restrictions on mutability and sharing, and provides a few different shared variable types. The closest to `IORef` is a reference-counting box containing an atomically modifiable pointer, shown in Listing 9. Threads can modify the pointer by cloning the shared `Arc` value, extracting the inner `AtomicPtr`, and updating the value inside. All mutation operations take as a parameter the type of memory consistency to enforce, which we shall discuss shortly.

Shared references under mutual exclusion An `MVar` is a mutable location in memory with two possible states: *full*, holding a Haskell value, and *empty*, holding no value. An `MVar` can be created in either state. The API is shown in Listing 10.

Writing to a full `MVar` blocks until it is empty, and reading or taking from an empty `MVar` blocks until it is full. There are also non-blocking functions which return an indication of success. The blocking behaviour of `MVar`s means that computations can become deadlocked. For example, deadlock occurs if every thread tries to take from the same `MVar`, with no threads writing to it.

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

putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a
takeMVar     :: MVar a -> IO a

tryPutMVar   :: MVar a -> a -> IO Bool
tryReadMVar  :: MVar a -> IO (Maybe a)
tryTakeMVar  :: MVar a -> IO (Maybe a)
```

Listing 10: Mutual exclusion in Haskell.

```
Semaphore sem = new Semaphore(/* initial quantity */);

/* from another thread */
sem.acquire(/* quantity */);
/* ... */
sem.release(/* quantity */);
```

Listing 11: Mutual exclusion in Java.

Java does not provide an exact analogue of `MVar`, but it does provide semaphores (Dijkstra 1965), shown in Listing 11, which can be used to control access to a shared resource. Furthermore, every Java object has an associated lock, which can be used to enforce mutual exclusion.

The Rust `Mutex` type, shown in Listing 12, is more like the Haskell `MVar` type. It does not merely function as a lock but also guards a reference. Locks are released when the unlocked value falls out of scope, ensuring that a thread cannot lock a mutex and terminate without unlocking it. There is also a non-blocking `Mutex::try_lock` function. There is no way to explicitly lock an unlocked mutex.

Memory model `IORef` operations may appear to be re-ordered, depending on the memory model of the underlying processor. The documentation has this to say:

```
let shared = Arc::new(Mutex::new(/* initial value */));

let shared_clone = shared.clone();
let thread = thread::spawn(move || {
  let mut unlocked = shared_clone.lock();
  /* ... */
});
```

Listing 12: Mutual exclusion in Rust.

```haskell
atomicWriteIORef  :: IORef a -> a -> IO ()
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
```

Listing 13: Atomic operations in Haskell.

```java
public volatile int sequentiallyConsistent = 0;
```

Listing 14: Atomic operations in Java.

> In a concurrent program, `IORef` operations may appear out-of-order to another thread, depending on the memory model of the underlying processor architecture. For example, on x86, loads can move ahead of stores.
> The implementation is required to ensure that reordering of memory operations cannot cause type-correct code to go wrong. In particular, when inspecting the value read from an `IORef`, the memory writes that created that value must have occurred from the point of view of the current thread.
> (The GHC Developers 2017b)

Many non-`IORef` operations are *synchronised*, and act as a *barrier* to re-ordering. Such operations include reading from or writing to an `MVar`, executing a software transaction (§2.4), and throwing an asynchronous exception (§2.5). There are also synchronised `IORef` operations, shown in Listing 13. In our work, we support the Total Store Order (TSO) (Owens, Sarkar, and Sewell 2009) and Partial Store Order (PSO) (SPARC International 1992) models (§5.4).

Java allows specifying how individual variables should be synchronised. Listing 14 shows a `volatile` integer. Operations on normal variables may appear out-of-order to different threads, however any operations on a `volatile` variable will be in-order.

As we saw on page 27, Rust operations which mutate atomic values specify the desired memory consistency. The weakest is `Relaxed`, which imposes no constraints, and the strongest is `SeqCst`, which imposes sequential consistency.

Sequential consistency While relaxed memory models are used for performance, research suggests that in languages which statically distinguish between shared and thread-local state such as Haskell, sequential consistency can be imposed for all shared state with little overhead (Vollmer et al. 2017). If implemented in Haskell, this policy would greatly simplify correct use of `IORef`s.

```
readForCAS :: IORef a -> IO (Ticket a)
peekTicket :: Ticket a -> a
casIORef   :: IORef a -> Ticket a -> a -> IO (Bool, Ticket a)
```

Listing 15: Compare-and-swap in Haskell.

```
private AtomicInteger count = new AtomicInteger(0);

public void increment() {
  count.incrementAndGet();
}
```

Listing 16: Compare-and-swap in Java.

Compare-and-swap Modern processor architectures provide an atomic *compare-and-swap* instruction, which is typically used in implementing lock-free algorithms (Dice, Hendler, and Mirsky 2013). The atomic-primops package (R. R. Newton 2017) provides a model of this instruction.

Listing 15 shows compare-and-swap in Haskell. A `Ticket` is a proof that a value has been observed inside an `IORef` at some prior point. Given this proof, the programmer can efficiently and atomically change the value inside the `IORef` later if it has not been modified. The `casIORef` function is partially synchronised, acting as a barrier to re-ordering of operations on that particular `IORef`, but not constraining other operations.

Java provides an "atomic" variant of each primitive type. These atomic types support compare-and-swap. Listing 16 shows the `AtomicInteger` type, an atomically modifiable 32-bit signed integer.

The Rust atomic types provide compare-exchange, shown in Listing 17, in addition to compare-and-swap. Compare-exchange differs from compare-and-swap in that the programmer specifies the desired memory consistency on failure. Specifying a weaker memory consistency on failure may improve performance in some cases, as synchronisation is expensive.

```
ptr.compare_and_swap(other, another, Ordering::SeqCst);
ptr.compare_exchange(other, another, Ordering::SeqCst, Ordering::Relaxed);
```

Listing 17: Compare-and-swap in Rust.

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

Listing 18: Transactional variables in Haskell.

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

Listing 19: Aborting and retrying transactions in Haskell.

2.4. Software Transactional Memory

Shared variables are nice, until we need more than one. As we can only claim one `MVar` atomically (or write to one `IORef` atomically), it seems we need to introduce additional synchronisation. This is unwieldy and prone to bugs. *Software transactional memory* (STM) (Harris et al. 2005; Shavit and Touitou 1995) is the solution. STM is based on the idea of atomic *transactions*. A transaction consists of one or more operations over a collection of *transaction variables*, where a transaction may be aborted part-way through, with all its effects rolled back. Arbitrary effects are not permitted, which Haskell enforces by having a distinct type for STM actions.

Neither Java nor Rust provide an STM implementation in their standard libraries, but there are third-party implementations. However, as Java and Rust are impure, these libraries cannot prevent the programmer from performing arbitrary effects inside a transaction. These STM library implementations provide atomic transactions for specified operations, but they *cannot* provide the same guarantees as STM in Haskell.

Transactional variables The `TVar` type is yet another type of shared variable, but with the difference that operating on them has a transactional effect. The API is shown in Listing 18.

Transactions are atomic, so all reads will see a consistent state and, in the presence of writes, intermediate states cannot be observed by another thread.

Aborting and retrying If we read a `TVar` and do not like the value it has, the transaction can be aborted. The thread will then block until any of the referenced `TVars` have been mutated. We can also try executing a transaction, and do something else if it retries, as shown in Listing 19.

```
atomically :: STM a -> IO a
```

Listing 20: Executing transactions in Haskell.

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
throw :: Exception e => e -> IO a
```

Listing 21: Exceptions in Haskell.

Executing transactions Transactions compose. We can take small transactions and build bigger transactions from them, and the whole is still executed atomically, as shown in Listing 20.

This means we can do complex state operations involving multiple shared variables without worrying about atomicity. However, using STM requires the program to be structured in a way which separates state modifications from other IO operations. Furthermore, due to how transactions are aborted and restarted when a conflict occurs, large transactions can be slow (Le and Fluet 2015).

2.5. Exceptions

Exceptions are a way to bail out of a computation early. Exceptions can be explicitly thrown within a single thread, these are *synchronous* exceptions, or thrown from one thread to another, these are *asynchronous* exceptions.

Throwing and catching The basic functions for dealing with exceptions are throwing and catching. The API is shown in Listing 21.

Throwing an exception causes the computation to jump back to the nearest enclosing suitable exception handler. If there is none, the thread terminates. Haskell exceptions belong to a typeclass, rather than having a specific type, so different `catch` functions can be nested, to handle different types of exception.

In addition to Haskell-style exceptions, Java supports *checked exceptions*, shown in List-

```
public void createFile(String path, String text) throws IOException {
  FileWriter writer = new FileWriter(path, true);
  writer.write(text);
  writer.close();
}
```

Listing 22: Checked exceptions in Java.

```
let result = panic::catch_unwind(|| {
  panic!("oh no!");
});
```

Listing 23: Panics in Rust.

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

Listing 24: Asynchronous exceptions in Haskell.

ing 22. If a method can throw (or propagate) a checked exception, this appears in the type signature. Checked exceptions statically enforce exception handling, but are often regarded as cumbersome. The Haskell type system has no equivalent of checked exceptions. If a Haskell programmer wants something like a checked exception, they use a type such as `Either` to indicate success or failure.

Rust does not really have exceptions. The `panic` function, shown in Listing 23, raises an error which, if uncaught, kills the current thread. The `catch_unwind` function can be used to execute a closure and recover from a panic, but it is not guaranteed to catch all panics (The Rust Developers 2018), making panics unsuitable as a general control-flow mechanism. The typical Rust approach is, like Haskell, to return a type indicating success or failure.

In addition to *synchronous* exceptions, Haskell has *asynchronous* exceptions, shown in Listing 24, which can be thrown to another thread. These functions block until the target thread is in an appropriate state to receive the exception. Asynchronous exceptions can be caught with `catch`, just like synchronous exceptions thrown with `throw`.

The Java `Thread.stop` method is similar to `killThread`, but is considered a bad idea and deprecated, as it causes the target thread to immediately release any locks it holds (Oracle 2017). The preferred approach is the `Thread.interrupt` method, which will either throw an exception or set a flag, depending on what the target thread is doing. For example, if the target thread is blocked inside a `Thread.sleep` call, it will receive an `InterruptedException`. Rust does not provide any way to tell a thread to terminate.

Masking A thread has a masking state, which can be used to block asynchronous exceptions thrown from other threads. There are three masking states: (1) *unmasked*, in which a thread can have exceptions thrown to it; (2) *interruptible*, in which a thread can only have exceptions thrown to it if it is blocked; and (3) *uninterruptible*, in which a thread cannot have exceptions thrown to it.

```
forkIOWithUnmask    :: ((forall a. IO a -> IO a) -> IO ()) -> IO ThreadId
forkOnWithUnmask    :: Int -> ((forall a. IO a -> IO a) -> IO ()) -> IO ThreadId

mask                :: ((forall a. IO a -> IO a) -> IO b) -> IO b
uninterruptibleMask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

Listing 25: Masking exceptions in Haskell.

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

Listing 26: STM exceptions in Haskell.

There are two functions to set the masking state. These each execute a computation in the new state, and pass it a function to run a subcomputation with the original masking state. When a thread is started, it inherits the masking state of its parent. As the parent may be masked, we can fork a thread with a function to run a subcomputation with exceptions unmasked. The API is shown in Listing 25.

Software transactional memory STM can also use exceptions, as shown in Listing 26. If an exception propagates uncaught to the top of a transaction, that transaction is aborted. The `orElse` function does not catch exceptions, and the `catchSTM` function does not catch retries.

```
import Control.Concurrent
import Control.Monad

main :: IO ()
main = forever $ do
  putStr "Enter a number of seconds: "
  s <- getLine                     -- 1
  forkIO (setReminder (read s))   -- 2

setReminder :: Int -> IO ()
setReminder s = do
  putStrLn ("Starting a " ++ show s ++ " second timer.")
  threadDelay (10^6 * s)         -- 3
  putStrLn "Time is up!\BEL"     -- 4
```

Listing 27: A simple alarm program. Adapted from Marlow 2013.

```
main :: IO ()
main = loop where
  loop = do
    putStr "Enter a number of seconds, or \"exit\": "
    s <- getLine
    if s == "exit"
      then pure ()
      else do
        forkIO (setReminder (read s))
        loop
```

Listing 28: A simple alarm program, with an exit instruction.

2.6. Example Program

Listing 27 shows a simple program which prompts the user for a number of seconds and prints a message, as well as ringing the terminal bell, after that time. This is a concurrent program. The user can keep entering new delays before old ones have elapsed. Execution proceeds as follows:

1. Get a number of seconds from the user.
2. Fork a thread to execute the `setReminder` function, and return to the prompt.
3. The new thread delays for the given number of seconds.
4. The new thread prints a message and sound the bell.

We can extend this program to allow the user to type “exit” to quit. Listing 28 shows the new `main` function. This program is similar to the original but, rather than using `forever`, we use a custom recursive function; we also only loop in the case where the input is not “exit.”

This program now demonstrates an important property of Haskell threading. The user can quit even if there are reminder threads still running. All Haskell threads terminate when the main thread does, regardless of what they are doing. Haskell provides the simplest behaviour, leaving it to libraries to implement higher-level behaviour using these building blocks.

Shared state We can modify our program to only quit when every reminder is done. To achieve this, we need to know if there are any reminders outstanding. One way to do this is to give every reminder thread an `MVar`, which we write to when done. Listing 29 shows the new `main` function.

```haskell
main :: IO ()
main = loop [] where
  loop vars = do
    putStr "Enter a number of seconds, or \"exit\": "
    s <- getLine
    if s == "exit"
      then mapM_ readMVar vars
      else do
        var <- newEmptyMVar
        forkIO $ do
          setReminder (read s)
          putMVar var ()
        loop (var:vars)
```

Listing 29: A simple alarm program, keeping a list of alarms.

```haskell
main :: IO ()
main = loop =<< newMVar 0 where
  loop var = do
    putStr "Enter a number of seconds, or \"exit\": "
    s <- getLine
    if s == "exit"
      then wait var
      else do
        modifyMVar_ var (+1)
        forkIO $ do
          setReminder (read s)
          modifyMVar_ var (-1)
        loop var
  wait var =
    c <- readMVar var
    if c == 0 then pure () else wait var
```

Listing 30: A simple alarm program, keeping a counter of alarms.

For each reminder, we create an empty `MVar`. A reminder thread fills its `MVar` when done. On exit, each `MVar` is read from, which will block if the `MVar` is still empty.

So now the main thread will block until every reminder is done, however this approach is not satisfactory. Our `MVar` list gains one element each time we set a reminder, so our program has linear space usage. We can instead use a shared counter, and wait for the count to be zero before terminating. Listing 30 shows the new `main` function.

So now, rather than have a constantly growing list, we just have a single `MVar`. When a new reminder is created the value in the `MVar` is incremented. When a reminder terminates the value in the `MVar` is decremented. However, this program has two flaws. Firstly, `modifyMVar_` is not atomic: if two threads are updating the counter at the same

```
main :: IO ()
main = loop =<< newTVarIO 0 where
  loop var = do
    putStr "Enter a number of seconds, or \"exit\": "
    s <- getLine
    if s == "exit"
      then wait var
      else do
        atomically (modifyTVar var (+1))
        forkIO $ do
          setReminder (read s)
          atomically (modifyTVar var (-1))
        loop var
  wait var = atomically $ do
    c <- readTVar var
    if c == 0 then pure () else retry
```

Listing 31: A simple alarm program, keeping an atomic counter of alarms.

time, one may undo the other's effect. Secondly, we wait for all reminder threads to be done by looping until a condition holds, which is inefficient.

Software transactional memory We can solve the two problems with the `MVar`-counter approach using STM instead. Transactions are atomic, so we can modify the counter atomically. When aborted, a transaction blocks until any referenced variables are updated, which is more efficient than repeatedly checking. Listing 31 shows the new `main` function.

It may not be obvious why reading a `TVar` and aborting the transaction is more efficient than reading an `MVar` and looping. But consider the scheduling behaviour. The `TVar` approach will block until another thread writes to it. However, the `MVar` approach will not block at all. So in the `MVar` case, the thread could be scheduled multiple times in a row, even though this is a waste of time.

This is the final version of our program.

2.7. Summary

Going forward, the reader should keep in mind:

- GHC uses a green threading model. Multiple Haskell threads are multiplexed onto a smaller number of operating system threads. Two Haskell threads can execute in parallel if they are mapped to different operating system threads (§2.2).
- Haskell threads may explicitly yield control to another thread, or block themselves until some delay has elapsed (§2.2).
- An `IORef` is a mutable reference, used for communication between threads. `IORef` operations come in two kinds: synchronised and unsynchronised. Depending on the memory model of the processor the program is running on, unsynchronised operations may appear to happen out-of-order (§2.3).
- An `MVar` is another kind of mutable reference, but only has synchronised operations. An `MVar` may be full or empty: attempting to write to a full `MVar`, or attempting to read from an empty `MVar`, blocks. `MVars` are used to implement mutual exclusion (§2.3).
- A `TVar` is yet another kind of mutable reference, used to implement software transactional memory. Unlike `IORef` and `MVar` operations, `TVar` operations can be composed, and the whole executed atomically. Transactions do not permit arbitrary effects, only effects on `TVars` (§2.4).
- Haskell has exceptions. Like Java, arbitrary exception types can be created. When an exception is thrown, control jumps back to the nearest enclosing suitable exception handler. If there is no such handler, the thread is terminated (§2.5).
- So-called asynchronous exceptions can be thrown between threads. An asynchronous exception is raised in the target thread like a normal exception: control jumps back to a suitable exception handler, or kills the thread (§2.5).
- A thread can prevent the delivery of asynchronous exceptions by changing its masking state. There are three states: (1) unmasked, allowing asynchronous exceptions to be delivered; (2) masked interruptible, only allowing an asynchronous exception to be delivered when the thread is blocked; and (3) masked uninterruptible, not allowing asynchronous exceptions at all. Throwing an asynchronous exception to a thread not in a suitable state to receive it blocks (§2.5).

We assume familiarity with Concurrent Haskell throughout the rest of the thesis.

Chapter 3

Concurrency Testing by Controlled Scheduling

Testing concurrent programs cannot be done with conventional techniques. The non-determinism of scheduling means that a test may produce different results in different executions. In this chapter we give an introduction to concurrency testing through *controlled scheduling*, which addresses this problem. Controlled scheduling is the foundation upon which we build our work. We first give a high-level overview (§3.1), then discuss specific implementation approaches, both complete (§3.2) and incomplete (§3.3). We then discuss two tools for concurrency testing in functional languages (§3.4).

This chapter is presented in a different style to Chapter 2. We are now discussing ideas and approaches, rather than the specifics of particular programming languages.

3.1. Controlled Scheduling

With a controlled scheduling technique, execution of a program is serialised and the controlling scheduler drives the program. Program schedules are either explored *systematically* (Flanagan and Godefroid 2005; Madanlal Musuvathi and Qadeer 2007) (often called 'systematic concurrency testing' or 'SCT') or randomly (Thomson, Donaldson, and Betts 2016). Non-controlled methods do not provide their own scheduler, and instead use delays and priorities to affect execution (Yu et al. 2012). Controlled scheduling techniques are attractive because of their ability to record and replay program executions.

Controlled scheduling can be implemented by overriding the concurrency primitives of the language (Walker and Runciman 2015); by instrumenting the source program (Claessen, Palka, et al. 2009); or by instrumenting the compiled program (Madan Musuvathi and Qadeer 2007; Yu et al. 2012). Systematic techniques may be *complete*: able to find all distinct results of a program. Random techniques typically cannot ensure that all distinct results are found—cannot be complete—and are usually run for some predetermined number of program executions.

Typically we require that all possible executions are *terminating*: leading to successful completion or a failure state such as deadlock after a finite number of scheduling decisions. Another common requirement is that the number of possible executions is *finite*: forbidding finite but arbitrarily long executions, as can be created with constructs such as spinlocks (Siberschatz and Galvin 1993). We can sacrifice completeness to do away with these requirements, as we shall see in Section 3.3.

3.2. Dynamic Partial-order Reduction

Dynamic partial-order reduction (DPOR) (Flanagan and Godefroid 2005; Godefroid 1996) is a *complete* approach to SCT. It is based on the insight that, when constructing schedules, we only need to consider different orderings of a pair of actions if the order in which they are performed could affect the result of the program. We call this relation between actions the *dependency relation*.

Definition (Dependency Relation (Flanagan and Godefroid 2005))
Let $\mathcal{T}$ be the set of transitions in a concurrent system. A binary, reflexive, and symmetric relation $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{T}$ is a valid dependency relation iff, for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin \mathcal{D}$ (t_1 and t_2 are independent) 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 (unblock) or disable (block) each other, and enabled independent transitions commute. When implementing DPOR, we typically identify a sufficient and easy-to-compute condition for dependency, rather than work with this semantics-based definition directly.

Typically, the presentation of algorithms assumes a simple core concurrent language of just reads and writes. This gives rise to a relation such as: x and y are dependent if and only if they are actions in the same thread, or they are actions involving the same variable where at least one is a write. We can express this dependency relation like so:

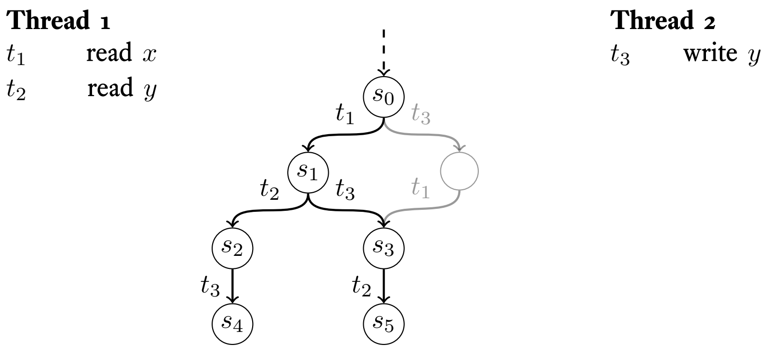


Figure 1: How DPOR prunes the space of schedules. Transition t_3 is pruned in state s_0 because it is independent with transition t_1. Adapted from Coons, Madan Musuvathi, and McKinley 2013.

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

The notation $x \not\leftrightarrow y$ is read as "x and y are dependent." This choice of notation would suggest $\leftrightarrow$ for independence, but that does not seem to be in common use.

Figure 1 shows an example of DPOR in action. There are two threads: thread 1 performs reads from two variables x and y, and thread 2 writes to y. As `read x` is independent from `write y`, DPOR prunes one ordering of those actions.

Our dependency relation for Haskell (§5.6) is rather more complex, as there are more actions than just reads and writes. Additionally, a Haskell program terminates when the main thread terminates, which complicates matters further. A naïve implementation of this, imposing a dependency between the final action of the main thread and everything else, leads to too many executions being tried to be of practical use. We discuss these issues further in Section 5.6.

3.2.1. Total and Partial Orders

Characterising the execution of a concurrent program by the ordering of its dependent actions gives us a *partial* order over the actions in the entire program. An execution trace is just one possible *total* order, a refinement of the constraining partial order. We call the equivalence class of total orders corresponding to the same partial order a *Mazurkiewicz trace* (Mazurkiewicz 1986). The goal of partial-order reduction, then, is to only try one total order for each distinct Mazurkiewicz trace, by intelligently making scheduling decisions to permute the order of dependent actions.

DPOR is so called because it gathers information about the dependencies between threads dynamically at run-time, to avoid the imprecision of static analyses (Flanagan and Godefroid 2005). It works by executing the program until completion, making arbitrary choices to resolve scheduling nondeterminism, dynamically collecting information about how threads have behaved during this specific execution. This execution trace is then examined to identify places where alternative scheduling decisions need to be explored because they might lead to other executions which correspond to a different partial-order. The algorithm repeats until all such alternatives have been explored and no new ones are found. So DPOR only works if all executions are terminating and the number of distinct executions is finite. DPOR is complete. When it terminates, all distinct states of the program will have been explored.

3.2.2. Relaxed Memory Models

In the name of performance, modern processors implement memory models that are weaker than sequential consistency (Lamport 1979) by using optimisations such as speculative execution, buffering, and caching. Unlike sequential consistency, where a concurrent program behaves as a simple interleaving of atomic thread actions, relaxed memory models can be more complex, making program analysis and debugging difficult. For example, under Total Store Order (TSO), which x86 processors use (Owens, Sarkar, and Sewell 2009), a read and a write in the same thread to different memory locations may be re-ordered. Under Partial Store Order (PSO), a relaxation of TSO (SPARC International 1992), two writes in the same thread, but to different memory locations, may also be reordered.

SCT techniques assume that there is only one source of nondeterminism: the scheduler. If a second source is added, such as when writes become visible, adapting existing algorithms is difficult. We can reconcile relaxed memory nondeterminism and scheduler nondeterminism using a simple buffering technique (Zhang, Kusano, and Wang 2015):

- Under TSO, each thread has a queue of buffered writes.
- Under PSO, each thread has a queue of buffered writes for each shared variable.

When reading, a thread reads its most recently buffered write. If a thread has no writes buffered to that variable, it reads the most recent globally visible write. A buffered write is only visible to the thread which made it. Buffered writes are committed to the globally visible memory nondeterministically. To model this, we can introduce one additional *phantom thread* for each nonempty buffer. When scheduled, a phantom thread commits the oldest write from its buffer. By using phantom threads, the two sources of nondeterminism are unified, and existing algorithms just work (Zhang, Kusano, and Wang 2015).

3.2.3. Maximal Causality Reduction

Maximal causality reduction (MCR) (J. Huang 2015; S. Huang and J. Huang 2017) is an alternative to DPOR which explores a provably minimal number of executions. Consider these three threads:

```
p: write x        q: write x       r: read x
```

All pairs of actions are dependent, and so DPOR would explore all six interleavings: `pqr`, `prq`, `qpr`, `qrp`, `rpq`, `rqp`. However, if we consider which write is read by thread `r`, many of these interleavings are equivalent. For example, `pqr` results in the same value being read as `qrp`. In fact we only need to explore half of the interleavings to find all the distinct values read. Program execution is driven by what values different threads read. An unread write changes nothing. So ideally we would only try a schedule if it leads to at least one distinct value being read.

The MCR algorithm is similar in outline to DPOR. It performs an execution, resolving scheduling nondeterminism arbitrarily, and gathers a trace including information about the thread communication. It then uses this trace to compute new schedule prefixes. The difference from DPOR is that these schedule prefixes ensure that at least one read produces a previously unseen value. MCR uses the trace to compute a model of program behaviour as a set of quantifier-free first-order logical formulae. These formulae can then be augmented with a state-change requirement and given to an SMT solver (De Moura and Bjørner 2011), such as z3 (De Moura and Bjørner 2008), to produce new schedule prefixes. When executed on benchmark programs, MCR outperforms DPOR by orders of magnitude (S. Huang and J. Huang 2017).

MCR imposes one additional restriction which makes it tricky for Haskell. MCR requires a concurrency model to be *locally deterministic* (J. Huang 2015). Only the previous actions of a thread and values read from shared variables, and not actions of other threads, determine the next action of the thread. This is not the case for Haskell, where one thread may kill another by throwing an exception to it. However, it may be possible to encode Haskell exceptions in an MCR-friendly way by giving each thread an exception variable, and inserting reads to this variable before every normal action. Even after this modification some difficulty remains, as even blocked threads may be interrupted by exceptions in Haskell.

Like DPOR, MCR can be extended to support the relaxed memory models TSO and PSO (S. Huang and J. Huang 2016).

3.3. Schedule Bounding

Schedule bounding (Emmi, Qadeer, and Rakamarić 2011; Madanlal Musuvathi and Qadeer 2008; Madanlal Musuvathi and Qadeer 2007) is an *incomplete* approach to concurrency testing. A *bound function* is defined which associates a sequence of scheduling decisions with some value of a type that has a total order, such as the integers. This function is monotonically increasing: if some sequence has an associated value of n, all its prefixes will have an associated value of at most n. This value n is limited by some predetermined bound. Testing proceeds by executing all schedules within the bound.

A common schedule bounding approach is *pre-emption bounding* (Madanlal Musuvathi and Qadeer 2007), which limits the number of pre-emptive context switches. Empirical evidence shows that small bounds, and small numbers of threads, are effective for finding many real-world bugs (Thomson, Donaldson, and Betts 2014).

Another common approach is *fair bounding* (Madanlal Musuvathi and Qadeer 2008), which bounds the difference between how many times any two threads may explicitly yield. This prevents infinitely long executions when using constructs such as spinlocks, which loop until some condition holds, yielding on every iteration it does not.

Bound functions can be combined, where a sequence of scheduling decisions is outside the combined bound if it is outside any of the constituent bounds.

Schedule bounding traditionally refers to trying only those schedules with a bound value equal to a fixed parameter. A variant is *iterative* bounding, where the parameter is gradually increased (Madanlal Musuvathi and Qadeer 2007). Another variant is where an inequality, rather than an equality, is used. This variant explores the same schedules as iterative bounding, but does not impose the same ordering.

3.3.1. Integration with DPOR

Schedule bounding can be combined with DPOR to produce a technique which is complete within its bound. The naïve way to integrate these techniques would be to first use partial-order techniques to prune the search space, and then to additionally filter things out with schedule bounding. However, this is unsound. As Figure 2 shows, this approach misses parts of the search space reachable within the bound. This is because

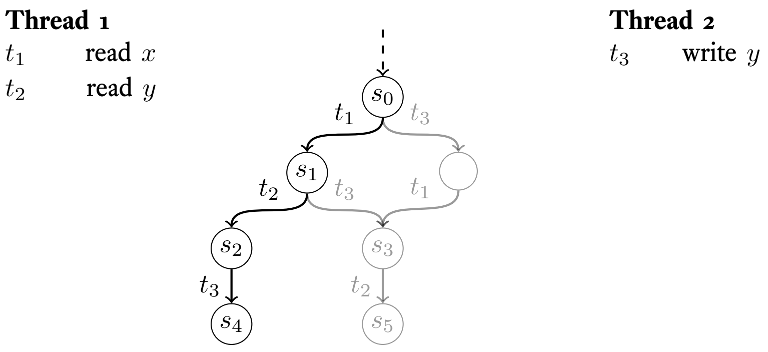


Figure 2: The naïve, unsound, way to combine DPOR with schedule bounding. Transition t_3 is pruned in state s_0 because it is independent with transition t_1. Transition t_3 may be pruned in state s_1 because it causes a pre-emption. If it is, the unique states s_3 and s_5 are never reached, despite being reachable within the bound from state s_0. Adapted from Coons, Madan Musuvathi, and McKinley 2013.

the introduction of the bound creates new dependencies between actions, which cannot be determined *a priori* (Coons, Madan Musuvathi, and McKinley 2013).

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 is sufficient to try all possibilities at the last context switch before a normal backtracking point (Coons, Madan Musuvathi, and McKinley 2013). This is because context switches influence the number of pre-emptions needed to reach a given program state, depending on which thread gets scheduled. So in Figure 2, the transition t_3 in state s_0 would be added as a conservative backtracking point, undoing the work of DPOR in that case. In practice the addition of backtracking points in this way tends not to greatly increase the search space (Coons, Madan Musuvathi, and McKinley 2013).

Listing 32 gives the bounded DPOR algorithm (Coons, Madan Musuvathi, and McKinley 2013). The Explore function is common to all bound functions: it takes a sequence of scheduling decisions, which is empty at first, and systematically explores states reachable from it. The Backtrack and Initialise functions are specific to the bound function: Backtrack adds scheduling decisions to the backtrack set for a state and introduces the necessary conservative backtracking points, and Initialise seeds the backtrack set with at least one scheduling decision. When execution of Explore$(\varnothing)$ terminates, all unique states reachable within the bound will have been explored.

```
function EXPLORE($S$)
    $s \leftarrow \text{final}(S)$
    # add backtrack points
    for $u \leftarrow Tid$ do
        for $v \leftarrow Tid, u \neq v$ do
            # find most recent dependent transition
            if $\exists i = \max(\{i \in \text{dom } S \mid S_i \nleftrightarrow \text{next}(s,\ u) \land S_i.tid = v\})$ then
                BACKTRACK($S,\ i,\ u$)
    # continue the search by exploring successor states
    INITIALISE($S$)
    $visited \leftarrow \varnothing$
    while $\exists u \in (\text{enabled}(s) \cap \text{backtrack}(s) \setminus visited)$ do
        add $u$ to $visited$
        if BV($S.next(s,\ u)) \leq c$ then
            EXPLORE($S.next(s,\ u)$)
```

Listing 32: The bounded DPOR algorithm. S is a sequence of scheduling decisions. The EXPLORE and INITIALISE functions are specific to each bound function. The BV function computes the bound value of a state. Adapted from Coons, Madan Musuvathi, and McKinley 2013.

3.4. In Functional Languages

In Erlang PULSE (Claessen, Palka, et al. 2009) is a controlled scheduler for Erlang programs which implements co-operative multi-tasking. An instrumentation process automatically modifies existing programs to call out to this scheduler. PULSE works by only allowing one of the concurrent processes to operate at a time, and makes scheduling decisions around effectful actions: such as a process receiving a message. It also allows interaction with uninstrumented functions, which are treated as atomic, allowing tested subsystems to be composed without exploring interleavings within the subsystem. PULSE scheduling decisions are made randomly, using a given seed, and a complete execution trace is returned. The trace can be rendered into a graphical form showing the interactions between threads to aid debugging. The authors report that the graphical traces often suggest potential race conditions not otherwise apparent to a human reader.

Procrastination (Sen 2008) is used to improve detection of bugs. First PULSE is used to produce an execution trace, which is then examined to find pairs of dependent actions, as in DPOR. Then, for each pair, execution proceeds with a random scheduler. When one of the actions in an identified dependent pair is encountered, the thread is instead paused until another thread is about to resolve the other action. The race is then randomly resolved and execution continues. Rather than exploring all partial orders, this approach is a probabilistic one, but it is guaranteed to explore only *racing* partial orders. This approach has an advantage in programs which have many non-racy partial orders, where randomly choosing between them does not reliably produce a bug. The authors report that improvements can result in new bugs being found, although in the cases where the procrastination was not necessary to find the bug, performance degrades (Arts, Hughes, Norell, et al. 2011) as one test with procrastination corresponds to multiple executions with different schedules.

Concuerror (Christakis, Gotovos, and Sagonas 2013) is another concurrency testing tool for Erlang. Like PULSE, Concuerror uses source instrumentation to insert calls to a scheduler process in the program under test. Unlike PULSE, Concuerror systematically explores the space of schedules by using iterative pre-emption bounding. Concuerror also uses *blocking avoidance* to prune redundant schedules. The insight here is that, if scheduling a thread will cause it to immediately block without updating any shared state, then there is no point in scheduling it. Concuerror does not use DPOR.

In Haskell The Concurrent Haskell Debugger (CHD) (Böttcher and Huch 2002) is a GUI-based controlled scheduler for Haskell programs. Like PULSE, CHD works by inserting blocking communication operations around concurrency actions to call out to the controlling scheduler. Unlike PULSE, this process is not automatic. The programmer must import the concurrency module provided by the CHD library, rather than the standard library. CHD does not implement its own scheduler. Rather, it presents a GUI to the user, allowing them to drive execution by clicking representations of threads. Furthermore, it allows the user to specify cases which should be automatically allowed to execute. CHD does not function with any GHC newer than version 5 (released between 2001 and 2003).

3.5. Summary

Going forward, the reader should keep in mind:

- Controlled scheduling techniques use a user-level scheduler to drive the execution of concurrent program. This can be done by overriding the concurrency primitives of the source language; instrumenting the program source code; or instrumenting the compiled program (§3.1).

- Systematic concurrency testing (SCT) is an umbrella term for a collection of techniques for exploring the behaviours of concurrent programs, through controlled scheduling (§3.1).
- Dynamic partial-order reduction (DPOR), which falls under the SCT umbrella, is a technique to discover all distinct states of a concurrent program. DPOR takes advantage of mutually commuting operations to reduce the space of schedules to explore (§3.2).
- Schedule bounding is a technique to reduce the space of schedules to explore by simply discarding any which exceed some chosen bound, such as the number of pre-emptive context switches. Schedule bounding will not, in general, find all distinct states of a concurrent program (§3.3).

We revisit DPOR and schedule bounding in Chapter 5, where we discuss our tool for testing concurrent Haskell programs. We revisit controlled scheduling more generally in Chapter 6, where we propose a new scheduling algorithm for exposing concurrency bugs.

Chapter 4

Property-based Testing

A common approach to testing in Haskell is to give properties about the code. Properties are functions with boolean results, expected to be true for all argument values. Property testing tools are used to generate input values and check that these properties hold, or display a counterexample if they do not. The popularity of property testing stems from the difficulty of writing good tests. In this chapter we give an overview of using property testing tools. We build on this background in Chapter 7, where we discuss a tool to *generate* properties for concurrency functions operating over shared state. We first give a general introduction to specifying and using properties as tests (§4.1), then discuss specific tools (§4.2). Finally, we summarise tools for generating properties (§4.3).

4.1. Properties as Tests

Property-based testing (Claessen and Hughes 2000), sometimes called *parameterised unit testing*, is an approach to testing where the programmer gives general laws (or properties) which should hold for all input values. For example, the property in Listing 33 says that a sort function should preserve length. Property-testing is unlike typical unit testing, which can be thought of as checking specific pairs of input–output values.

Often we do not want to check a property for arbitrary input values. Perhaps we know something about how the functions we are testing are expected to be used, or we are only interested in how they behave in a certain case. A simple way to enforce a precondition is to use logical implication. Implication is typically provided as part of the property

```
prop_sort_len xs = length xs == length (sort xs)
```

Listing 33: A property asserting that sorting preserves length.

```
prop_ord_insert1 x xs = ordered xs ==> ordered (insert x xs)
prop_ord_insert2 x    = forAll orderedList (\xs -> ordered (insert x xs))
```

Listing 34: Enforcing a precondition for a property.

language of a tool, rather than being a normal boolean function. This allows the tool to ensure that a desired number of generated inputs pass the precondition.

While implication is a useful technique, it can skew the input distribution. For example, in a property which uses implication to restrict the inputs to sorted lists, a naïve implementation would test more singleton lists than two-element lists: all singleton lists are ordered, but only 50% of two-element lists are. An alternative approach is to use a custom value generator. By only generating input values which satisfy the precondition, we can avoid skewing the distribution, and improve our confidence that the property does hold in general. Listing 34 shows the implication and generator function approaches.

In the absence of a programmer-supplied generator function, input values are generated in a type-directed process. A tool will usually provide a typeclass for generating values. This typeclass will have instances for most common types, but if a programmer wishes to have input values of other types, they will need to supply a suitable instance.

4.2. Property-based Testing Tools

Property-based testing tools mainly differ along two axes: the expressiveness of the property DSL, and the strategy for generating input values. Table 1 summarises the differences between five different tools for Haskell. Unlike our work in Chapter 7, none of these tools directly support concurrency. They may support properties with side-effects, but as they have no knowledge of concurrency, their ability to describe concurrency errors to the user is limited. We discuss them here to give an overview of property-based testing to the unfamiliar reader.

Input value generation Inputs can either be generated randomly or enumerated. Although simple, randomisation tends to work well in practice. QuickCheck (Claessen and Hughes 2000) is an example of a tool using random value generation. Alternatively, we may assume that there is some enumeration likely to expose useful counterexamples. SmallCheck (Runciman, Naylor, and Lindblad 2008) and LeanCheck (Braquehais 2017a) enumerate values in size order, on the assumption that most bugs are exhibited by simple counterexamples.

	QuickCheck	SmartCheck	SmallCheck	Lazy SmallCheck	LeanCheck
Input value generation					
random	●	●	○	○	○
enumerative	○	○	●	●	●
Property DSL					
existential	○	○	●	●	●
higher order	●	●	●	●	●
Output					
generalised counterexamples	○	●	○	◐	◐

Legend: ● Yes/Good. ○ No/Poor. ◐ Partial/Median.

Table 1: Summary of differences between property-based testing tools for Haskell. Adapted from Braquehais 2017b.

```
prop_gt_5 = exists (\x -> x > 5)
```

Listing 35: Using existential quantification in a property.

Simple counterexamples are more useful to the programmer than large ones, so random approaches must have an additional *shrinking* step, to try to reduce a counterexample to a local minimum. We use this idea of shrinking in Section 5.7 to simplify execution traces generated by randomised scheduling techniques such as PCT.

Property DSL A more expressive property language complicates implementation, but allows the programmer to say more. Two important types of property are *existential* properties and *higher-order* properties.

Existential properties, such as in Listing 35, allow the programmer to assert that some input exists for which the property holds. Existential properties are seemingly incompatible with a randomised tool, such as QuickCheck, because a random test value is unlikely to be a witness for a specific existential property. Existential properties are more commonly supported by enumerative tools.

Higher-order properties, such as in Listing 36, are properties where some of the inputs are functions. To test such a property requires the tool to be able to generate functions.

```
prop_map_fuse xs f g = map g (map f xs) == map (g . f) xs
```

Listing 36: Using higher-order functions in a property.

```
> check $ \xs -> nub xs == (xs::[Int])
*** Failed! Falsifiable (after 3 tests):
[0,0]

Generalization:
x:x:_
```

Listing 37: A generalised counterexample of an incorrect property.

Higher-order properties are invaluable in the testing of higher-order functions.

In the context of concurrency, another way in which a property could be *higher-order* is by taking an explicit schedule as an argument. A list of scheduling decisions cannot be generated up front, as the property-testing tool cannot know which threads are runnable. So instead we can generate a scheduler function, as in Ankuzik 2014. This is one possible way to implement random testing of concurrent programs.

Output How a tool presents its output is of great importance. Randomly generated counterexamples, such as those found by QuickCheck (Claessen and Hughes 2000), are often not minimal. Searching for a local minimum by shrinking randomly generated counterexamples before displaying them is a common approach. However, shrinking and enumeration are not the only ways to produce small counterexamples. Both SmartCheck (Pike 2014) and Lazy SmallCheck (Runciman, Naylor, and Lindblad 2008) can generalise counterexamples. LeanCheck (Braquehais 2017a) can generalise counterexamples when used with the Extrapolate (Braquehais and Runciman 2017a) tool. Generalising counterexamples directly can be more efficient than a shrinking process as in QuickCheck (Pike 2014). Furthermore, it is often possible to produce a generalisation which is simpler than any concrete counterexample. Listing 37 shows such a generalised counterexample. The property here fails for lists which contain duplicates, the concrete value is unimportant.

None of the tools in this section support concurrency directly, so even if they can be used to test concurrency properties, their ability to give useful output is limited.

Beyond Haskell Although this is a thesis using Haskell, a language particularly suited for property-based testing, the interest in property-based testing has spread to other languages. Property-based testing tools are available for at least Erlang (Arts, Hughes, Johansson, et al. 2006), Go (The Go Developers 2017b), Java (The JUnit Developers 2017; Holser 2018), .NET (The NUnit Developers 2017a; The NUnit Developers 2017b), and Python (HypothesisWorks 2018).

Although QuickCheck was arguably the first tool to popularise this style of testing, and did so in Haskell, it is increasingly gaining recognition by programmers of other languages as a good way to overcome the pitfalls and difficulties of traditional unit testing techniques.

4.3. Searching for Properties

Properties can be used as expressive and declarative test cases. However, coming up with properties can be difficult. To help the programmer, tools exist to discover properties. These tools are based on testing of examples, and so any properties found are merely conjectures supported by a finite amount of evidence. Despite that, such properties are surprisingly accurate in practice, and often lead to a deeper understanding of the program under test.

Testing QuickSpec (Claessen, Smallbone, and Hughes 2010; Smallbone et al. 2017) and Speculate (Braquehais and Runciman 2017b) are tools for Haskell which automatically discover equational laws of pure functions. Both are based on generating and testing candidate expressions. Speculate, unlike QuickSpec, can discover inequalities and conditional equations. Neither supports functions with effects or generating lambda-terms. We build on this approach in Chapter 7, to build a tool which discovers concurrency properties through generating and testing program fragments.

Listing 38 shows the properties Speculate finds when provided with the integers `0` and `1` and the functions `id`, `abs`, and `+`. QuickSpec discovers similar properties to Listing 38a, but not the inequalities and conditional equations in Listing 38b.

Machine learning The Daikon (Ernst et al. 2007) tool discovers *likely invariants* of C, C++, Java, and Perl programs. It observes variables in memory during the execution of a program, and applies machine learning techniques to discover properties that seem to hold. These properties may include: pre- and post-conditions of statements, and equational relationships between variables at a given program point and functions from a library. Daikon does not synthesise and test program terms, however. Daikon is provided with a grammar describing patterns of invariants, and reports which of these are observed to hold as the program executes. Properties found by Daikon correspond to

```
          id x == x
           x + 0 == x
     abs (abs x) == abs x
           x + y == y + x
     abs (x + x) == abs x + abs x
 abs (x + abs x) == x + abs x
 abs (1 + abs x) == 1 + abs x
     (x + y) + z == x + (y + z)
```

(a) Equational laws.

```
          x <= abs x                   x <= y ==> x <= abs y
          0 <= abs x               abs x <= y ==> x <= y
          x <= x + 1                abs x < y ==> x <  y
          x <= x + abs y               x <= 0 ==> x <= abs y
          x <= abs (x + x)         abs x <= y ==> 0 <= y
          x <= 1 + abs x            abs x < y ==> 1 <= y
          0 <= x + abs x               x == 1 ==> 1 == abs x
      x + y <= x + abs y                x < 0 ==> 1 <= abs x
abs (x + 1) <= 1 + abs x               y <= x ==> abs (x + abs y) == x + abs y
                                       x <= 0 ==>      x + abs x == 0
                                   abs x <= y ==>    abs (x + y) == x + y
                                   abs y <= x ==>    abs (x + y) == x + y
```

(b) Inequalities and conditional equations.

Listing 38: Properties of arithmetic, discovered by Speculate.

```
/* Thread 1 */               /* Thread 2 */
p = &A                       p = NULL;
if (p != NULL) {
  p->x += 10;
}
```

Listing 39: Two threads using a shared pointer.

assertions which could be inserted into the program, whereas the other tools described here discover properties based on the program API.

Concurrency testing A variant of the Daikon tool discovers likely invariants of concurrent C and C++ programs using code instrumentation and systematic concurrency testing techniques (Kusano, Chattopadhyay, and Wang 2015). The invariants it finds are so-called *transition invariants* that capture the relations amongst mutable state shared between threads.

Listing 39 shows two threads using a shared pointer. If Thread 2 executes `p = NULL` after Thread 1 checks that `p != NULL` but before it executes the assigment `p->x += 10`, then an error will occur. Correct executions of the program will produce the invariant `p == orig(p)` for that if-statement, meaning that `p` is unchanged. Buggy executions will not. The authors argue that examining discrepancies between invariants can lead to greater understanding of the software under test and diagnosis of errors.

The Determin tool (Burnim and Sen 2010) infers deterministic specifications for procedures which make use of internal parallelism. A program may have many such procedures. These specifications are in the form of a precondition and a postcondition over program states. If we use $P(s, \sigma)$ to denote the resulting program state after executing procedure P in an initial state s with a schedule σ, then specifications are of the form,

$$\forall s, s', \sigma, \sigma'.\ \text{Pre}\left(s, s'\right) \Rightarrow \text{Post}\left(P\left(s, \sigma\right), P\left(s', \sigma'\right)\right)$$

For example, if the precondition is $s = s'$ and the postcondition is $v = v'$, where v is some variable assigned to by P, then the overall specification can be read as "for all schedules σ from state s, the variable v gets the same value (if execution terminates)."

Example-driven property discovery The Bach (Smith, Ferns, and Albarghouthi 2017) tool uses a database of examples of input/output values from functions to synthesise properties using a Datalog-based oracle. As it is based on examples, it is not tied to any particular programming language. Bach could even be used to discover properties of hardware components! Properties are of the form $G \Rightarrow P$, where both G and P are conjunctions of equalities $f(x) = y$, where f is some function in the database, and x and y may be constants or variables. It uses a notion of *evidence* to decide whether an inferred property holds: negative evidence consists of counterexamples; positive evidence consists of witnesses.

Bach is unsuitable for discovering properties of concurrent or other nondeterministic programs, however. Its notion of negative evidence requires functions in its database to have at most one output for each distinct input

4.4. Summary

Going forward, the reader should keep in mind:

- Property-based testing, also called parameterised unit testing, is a style of testing which uses universally quantified predicates as test cases (§4.1).
- There are two approaches to generating parameter values for properties: random, and enumerative. Random value generation requires an additional shrinking step, to reduce counterexamples to a local minimum, whereas enumerative approaches will always discover the smallest counterexample (§4.2).
- The programmer does not necessarily need to come up with properties themselves. Property discovery tools take a program API and search for properties. These generated properties can be used to further program understanding (§4.3).

We briefly revisit property-based testing in Chapter 5 in a case study. We discuss property discovery more significantly in Chapter 7, where we present a tool to discover properties about the effects of concurrency functions.

Part II

Testing Concurrent Programs

Chapter 5

Déjà Fu: Haskell Concurrency Testing

> [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." (Pratchett 2001)

Specialised tools are necessary to test concurrent programs. In this chapter we present and evaluate Déjà Fu, our library for testing concurrency in Haskell. We give an overview of the tool (§5.1), including the scope of bugs we aim to detect, and present our abstraction over the GHC Haskell concurrency functionality (§5.2). We then show an example of a small logic puzzle which we can represent as a concurrent program (§5.3). We explain how programs using our abstraction are executed (§5.4), and give our semantic rules (§5.5). We explain how we test programs (§5.6) and how we present execution traces to the user (§5.7). We then argue the correctness of our approach (§5.8). We present three case studies (§5.9), and finally evaluate our results (§5.10).

This chapter is derived from our previous work Walker and Runciman 2015 and Walker 2016.

5.1. What is Déjà Fu?

Déjà Fu is a tool for testing concurrent Haskell programs. It works by providing a typeclass abstraction over the concurrency operations of interest, called `MonadConc`, and using an implementation of this class based on inspectable continuations to explore the nondeterministic behaviours of a program under test.

Listing 40 gives the type signature of the function at the heart of Déjà Fu. The first parameter is configuration, controlling how Déjà Fu explores the space of executions. The second is the concurrent program to test: `ConcT m` is Déjà Fu's implementation

```
runSCTWithSettings :: MonadConc m
  => Settings m a
  -> ConcT m a
  -> m [(Either Failure a, Trace)]
```

Listing 40: The core API of Déjà Fu.

of `MonadConc`, it is parameterised by some monad `m` which is used to provide mutable variables. Typically this `m` will be `IO`. The function returns a collection of tuples.

The outputs which Déjà Fu produces are pairs, where the first component is the result of the program and the second is the execution trace which led to it. A program may not complete successfully: for example, it may deadlock. If the program does complete successfully, the first component of the tuple will be a `Right` containing the actual value produced, otherwise it will be a `Left` and the `Failure` value will show what went wrong. A `Trace` is a list of scheduling decisions, with a summary of what the chosen thread did in that step of execution.

In outline, `runSCTWithSettings` works like so:

1. Repeat until there are no more executions:
   (a) Set up the initial state, where only the main thread exists
   (b) While the main thread exists, and there are runnable threads:
      i. Choose a thread
      ii. Run the next action in that thread
   (c) Add the result pair to the output

We produce different schedules by making step (b) stateful. Déjà Fu supports three ways to explore the behaviours of a concurrent program:

Randomly
: Threads are chosen using a uniform random distribution. Exploration terminates after a fixed number of executions.

Swarm
: Threads are chosen using a weighted random distribution. Exploration terminates after a fixed number of executions. We discuss this algorithm further in Chapter 6.

DPOR
: Uses a notion of "to-do sets," where each scheduling point has an associated set

of threads to try in future executions. Execution terminates when all such sets are empty.

During execution, at each scheduling point, if the previously chosen thread is still runnable, choose that; otherwise, take all runnable threads, choose one arbitrarily, and record the others in the to-do set for this scheduling point.

After execution, the trace is examined to find pairs of dependent actions: for each thread T, for each action in thread T, find the most recent dependent action (if one exists) in a different thread, and add T to the to-do set corresponding to that scheduling point if T is runnable; otherwise add all runnable threads.

There are some subtleties and additional optimisations, which we discuss further in Section 5.6.

5.1.1. What sort of bugs can Déjà Fu detect?

Déjà Fu produces a list of `(Either Failure a, Trace)` values, where the `a` is the result of a successful execution. Déjà Fu supports "online" filtering, removing uninteresting result pairs or execution traces as they are found. Configuration for this online filtering is passed to `runSCTWithSettings` via the `Settings` value. Déjà Fu tests use predicates on the final, filtered, list of pairs: these predicates can check anything, for example all results being `Rights` and equal. If an execution yields a `Left` value, we say that it has *failed*. Déjà Fu can detect two different types of failure:

Deadlocks
: Every thread is blocked.

Uncaught exceptions
: The main thread is killed by an exception.

There are two types of failure which Déjà Fu itself may raise:

Aborts
: All scheduling decisions are forbidden by schedule bounding.

Internal errors
: An internal invariant is broken. This indicates a bug in Déjà Fu.

Finally, there are two types of failure which can arise through improper use of Déjà Fu, due to weaknesses in the current API design.

For an example of a Déjà Fu predicate, see Listing 41. The `gives` predicate is used to check that a program produces exactly the desired set of results, failing if unexpected

```haskell
gives :: (Eq a, Show a) => [Either Failure a] -> Predicate a
gives expected = ProPredicate
    { pdiscard = \r  -> if r `elem` expected then Just DiscardTrace else Nothing
    , peval    = \xs -> go expected [] xs (defaultFail xs)
    }
  where
    go waitingFor alreadySeen ((x, _):xs) res
      -- If it's a result we're waiting for, move it to the
      -- @alreadySeen@ list and continue.
      | x `elem` waitingFor  = go (filter (/=x) waitingFor) (x:alreadySeen) xs res
      -- If it's a result we've already seen, continue.
      | x `elem` alreadySeen = go waitingFor alreadySeen xs res
      -- If it's not a result we expected, fail.
      | otherwise = res
    go [] _ [] res =
      res { _pass = True }
    go es _ [] res =
      res { _failureMsg = unlines (map (\e -> "Expected: " ++ show e) es) }

    defaultFail xs = Result
      { _pass       = False
      , _failures   = filter (\(r, _) -> r `notElem` expected) xs
      , _failureMsg = ""
      }
```

Listing 41: The `gives` predicate. The `pdiscard` component is used for online filtering, removing uninteresting traces as they are found. The `peval` component takes the final list of pairs and determines if the test passes or fails.

```
evalSigWithSeed :: (MonadConc m, Ord o)
  => (x -> ConcT m s)         -- ^ Create a new instance of the state.
  -> (s -> x -> ConcT m o)    -- ^ The observation to make.
  -> (s -> x -> ConcT m ()) -- ^ Perform some concurrent interference.
  -> (s -> ConcT m ())        -- ^ The expression to evaluate.
  -> x                        -- ^ Pure value used to initialise the state.
  -> m (Set (Maybe Failure, o))
```

Listing 42: The `evalSigWithSeed` function. Runs a concurrent program and returns a set of observations and possible failures.

results are present or expected results are not. The predicate reduces memory usage by using online filtering to discard the execution traces of expected results, as we are only interested in execution traces which yield *unexpected* results. There is nothing special about `gives`, it is just a normal function written using the Déjà Fu API. It is provided as an example of the sort of thing that Déjà Fu can be used to check.

Equivalence testing Déjà Fu supports testing the observational equivalence of two concurrent programs. Observational equivalence here means that the observable effects of the program on a distinguished piece of shared state, using some observation function provided by the programmer, are equal for both programs.

Listing 42 shows the heart of the equivalence testing. It is built on top of the usual `runSCTWithSettings` function, but encapsulates some concerns relevant to equivalence testing. In particular, `evalSigWithSeed` introduces concurrent interference, which is necessary to distinguish atomic from non-atomic operations in some cases. Every execution, including failing executions, yields an observation value. This is necessary because a failing execution may nevertheless modify the state before it fails. We build on top of this function a tool to generate a number of initial states and compare two programs for all of them. Two programs are observationally equivalent, up to the given interference and observation functions, if each initial state yields the same sets observations.

We build upon this further in Chapter 7 where we also discuss refinement, and how to *discover* rather than just *test* these properties.

Invariant testing A feature which Déjà Fu does *not* currently have, but which makes sense, is invariant testing. A function could be provided to register an arbitrary concurrency action as an invariant. As Déjà Fu drives the execution of the program under test, these invariants can be checked atomically. This would be similar to, but more general than, the GHC Haskell function `always :: STM Bool -> STM ()`, which registers an invariant to be checked at the end of STM transactions.

5.1.2. Scope

We aim to support most of the functionality of GHC's concurrency API. However, some operations are more trouble than they're worth: for example, operations which introduce additional sources of nondeterminism, or which unavoidably require support from the runtime system. In particular, we do not support:

- Operations to block a thread until a file descriptor becomes available, as this introduces an additional source of nondeterminism.
- Operations to query which capability (OS thread) a Haskell thread is running on, as this also introduces an additional source of nondeterminism.
- Automatically detecting if a thread is deadlocked on an `MVar` or `TVar` and throwing an exception to it, as we cannot detect this situation in general without support from the garbage collector.

We also do not yet support *bound threads*: a Haskell thread which will always run on the same, unique, OS thread. Bound threads are essential for using the foreign function interface (FFI) to call C libraries which use thread-local state, to ensure the Haskell thread always sees its state and never the state of another thread. We have a prototype implementation, which is planned to be included in the next major release of Déjà Fu[1].

Semantic departures In the functionality we do support, we model behaviour as close as reasonably possible to GHC. We make a few departures from the traditional semantics where there is good reason to do so:

- Runtime errors, such as pattern match failures, can be caught as exceptions inside `IO`. As there is no non-`IO` way to do the same, Déjà Fu cannot catch these errors.
- We do not model threads delaying, and implement any delays as yields. It is not clear how to incorporate time into the testing model.

Embedded `IO` actions Déjà Fu supports testing computations with embedded `IO` actions provided that the programmer ensures that the action is atomic; that it is deterministic when executed with a fixed schedule; and that it does not block on the action of another thread. Failing to meet any of these conditions may lead to incomplete testing.

[1] https://github.com/barrucadu/dejafu/issues/126

```haskell
class (Monad m, {- other constraints omitted -}) => MonadConc m where
  type MVar m :: * -> *
  -- other types omitted

  newEmptyMVar :: m (MVar m a)
  newEmptyMVar = newEmptyMVarN ""

  newEmptyMVarN :: String -> m (MVar m a)
  newEmptyMVarN _ = newEmptyMVar

  putMVar  :: MVar m a -> a -> m ()
  readMVar :: MVar m a -> m a
  takeMVar :: MVar m a -> m a
  -- other operations omitted
```

Listing 43: A fragment of the `MonadConc` typeclass.

```haskell
instance Monad n => MonadConc (ConcT m) where
  type MVar (ConcT m) = ModelMVar m
  -- other types omitted

  newEmptyMVarN n = toConc (ANewMVar n)

  putMVar  var a = toConc (\c -> APutMVar var a (c ()))
  readMVar var   = toConc (AReadMVar var)
  takeMVar var   = toConc (ATakeMVar var)
  -- other operations omitted
```

Listing 44: A fragment of the `MonadConc` testing implementation.

5.2. Abstracting over Concurrency

There are three ways of implementing a concurrency testing tool: overriding the concurrency primitives of the language; instrumenting the source program; or instrumenting the compiled program. We adopt the first approach in Déjà Fu. Haskell's typeclass machinery lets us specify an interface for concurrency, and to provide different concrete implementations. There is one implementation using the `IO` type and the standard functions; there is another using our own type, which we can inspect.

We call our typeclass `MonadConc`: monads which do concurrency. Listing 43 shows a fragment. To define an instance, the programmer supplies concrete types for the abstract types and implementations of all undefined operations. Some operations have default definitions: for example, there are two ways of constructing an empty `MVar`. One way takes a name, which is displayed in debugging information, the other does not. Each has a default definition in terms of the other, so the programmer must supply at least one.

Implementation The type for our testing implementation is called `ConcT m`, which is a monad that has access to mutable references provided by some monad `m`, which will typically be `IO`. Listing 44 shows a fragment of the instance of `MonadConc` for this type. Each concurrency operation is of the same form: we take the arguments and wrap them up inside a data structure whose final argument is a continuation, which is then converted into a `ConcT` value.

We represent a concurrent computation as a large value. We can inspect each step of the computation by looking at the data constructor used. We call these constructors *primitive actions*. With these actions we express the operations in the `MonadConc` class.

5.3. The n Prisoners Problem

> There are n prisoners in solitary cells. There's a central living room with one light bulb. No prisoner can see the light bulb from their own cell. Every day, the warden picks a prisoner equally at random, and that prisoner visits the living room. While there, the prisoner may toggle the bulb. The prisoner also has the option of asserting that all n prisoners have been to the living room. If this assertion is false, all n prisoners are shot. However, if true, all prisoners are set free. Thus, the assertion should only be made if the prisoner is 100% certain of its validity. The prisoners are allowed to get together one night in the courtyard, to discuss a plan. What plan should they agree on, so that eventually, someone will make a correct assertion?

We can express this puzzle as a concurrency problem: the warden is the scheduler, each prisoner is a thread, and when the program terminates every prisoner should have visited the living room. So if every thread (prisoner) is scheduled (taken to the room), the prisoners are successful. Déjà Fu can give us execution traces. So, given some way of setting up the prison, we can use Déjà Fu to execute it and then examine the returned traces to discover if the prisoners are successful.

5.3.1. The Probabilistic Solution

One school of thought says to just wait for $10n$ days, because by then it's unlikely that any prisoner has not visited the room. The chance that any one prisoner will have been consistently missed is $\left(1 - \frac{1}{n}\right)^{10n}$, which converges to $\frac{1}{e^{10}}$.

Listing 45a shows an implementation of this strategy, and Table 2 shows how the prisoners fare over 100 random executions. We see that the number of room visits grows a

Prisoners	1	2	3	4	5	6	7	8
Successes	100	100	100	100	100	100	100	100
Failures	0	0	0	0	0	0	0	0
Avg. Room Visits	2	18.35	31.92	43.52	55.88	67.37	77.05	90.40

Table 2: The behaviour of the probabilistic solution.

little faster than ten for each additional prisoner, this is because of a quirk of our implementation. We have nominated one prisoner to be the leader, who is the only prisoner able to declare that all have visited the room. So our implementation ends up waiting $10(n-1)$ days for the non-leaders to visit, and then however many days it takes for the leader to visit after that.

5.3.2. The Perfect Solution

Perhaps our prisoners are more cautious, and even a small chance of death is too much. They want to be certain of their success. A slow but simple strategy is for the prisoners, like in our probabiistic solution, to nominate a leader. Only the leader can declare to the warden that everyone has visited the room. Whenever a prisoner other than the leader visits the room, if this is their first time in the room with the light off, they turn it on, otherwise they do nothing. Whenever the leader enters the room, they turn the light off. When the leader has turned the light off $n-1$ times, they tell the warden that everyone has visited. Listing 45b shows an implementation of these behaviours.

We can satisfy ourselves that this solution works for all cases by using Déjà Fu's systematic concurrency testing functionality, which is a combination of dynamic partial-order reduction and schedule bounding. Table 3a shows how the number of schedules explored and average number of room visits grows as the number of prisoners increases. It does not scale well.

This algorithm is something of a worst-case for DPOR. Every thread is modifying the same shared state, so DPOR has to try every interleaving. Taking another look at our prisoners, we can see two things which a human would use to decide whether some schedules are redundant or not:

1. If we adopt any schedule other than alternating leader / non-leader, threads will block without doing anything. So we should alternate.
2. When a non-leader has completed their task, they will always yield. So we should never schedule a prisoner who will yield.

```
leader :: MonadConc m => Int -> TVar (STM m) Int -> m ()
leader numPrisoners days = atomically $ do
  numDays <- readTVar days
  when (numDays < (numPrisoners - 1) * 10) retry

notLeader :: MonadConc m => TVar (STM m) Int -> m ()
notLeader days = forever $ atomically (modifyTVar days (+1))

prison :: MonadConc m => Int -> m ()
prison numPrisoners = do
  days <- atomically (newTVar 0)
  for_ [1..numPrisoners-1] (\_ -> fork (notLeader days))
  leader numPrisoners days
```

(a) The probabilistic solution: just wait a long time and gamble.

```
data Light = IsOn | IsOff

leader :: MonadConc m => Int -> TVar (STM m) Light -> m ()
leader numPrisoners light = go 0 where
  go counter = do
    counter' <- atomically $ do
      state <- readTVar light
      case state of
        IsOn -> do
          writeTVar light IsOff
          pure (counter + 1)
        IsOff -> retry
    when (counter' < prisoners - 1)
      (go counter')

notLeader :: MonadConc m => TVar (STM m) Light -> m ()
notLeader light = do
  atomically $ do
    state <- readTVar light
    case state of
      IsOn  -> retry
      IsOff -> writeTVar light IsOn
  forever yield

prison :: MonadConc m => Int -> m ()
prison numPrisoners = do
  light <- atomically (newTVar IsOff)
  for_ [1..numPrisoners-1] (\_ -> fork (notLeader light))
  leader numPrisoners light
```

(b) The perfect solution: nominate a leader, who waits until they are certain that everyone has been in the room.

Listing 45: Two solutions for the n prisoners problem.

Prisoners	1	2	3
Schedules	1	5	2035
Avg. Room Visits	2	7	133

(a) Using Déjà Fu's default schedule bounds.

Prisoners	1	2	3	4	5	6
Schedules	1	1	4	48	1536	122880
Avg. Room Visits	2	4	7.5	11.5	16	21

(b) Using a custom fair bound to prevent yields.

Table 3: How the number of schedules grows with increasing prisoner numbers.

Déjà Fu cannot make use of (1). It would be possible to implement this optimisation if Déjà Fu were able to compare values inside `TVars`, as we would be able to check if any `TVar` read during a blocked transaction has a different value: if none do, the transaction will just block again. This would require a new primitive action, as we cannot do that without putting an `Eq` constraint on `writeTVar`. We get part of the way there, though. If a thread is blocked in a transaction, Déjà Fu only tries executing it again when at least one of the `TVars` it reads from has been written to, which is similar to the version numbering scheme commonly used in STM implementations (Shavit and Touitou 1995).

Déjà Fu can make use of (2). Déjà Fu already bounds the maximum number of times a thread can yield, so that it can test constructs like spinlocks. This is called *fair bounding*. The default bound is five, but if we set it to zero Déjà Fu will never schedule a thread which is going to yield. Table 3b shows how the number of schedules explored and average number of room visits grow with this change.

This is better, but still scales poorly. The program is still a bad case for DPOR. This is probably as good as we can do without adding some extra primitives to Déjà Fu to optimise the case where we have an `Eq` instance available, or by using an alternative systematic testing algorithm. In Chapter 6 we will discuss an alternative *incomplete* approach to this, and other, concurrency problems.

It's not the end for DPOR, however. Empirical studies (Thomson, Donaldson, and Betts 2014) have found that many concurrency bugs can be exhibited with only two or three threads. Furthermore, most real-world concurrent programs do not have every single thread operating on the same bit of shared state. So in practice, we will tend not to see this exponential growth in schedules tried.

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

instance Functor (ConcT n) where
  fmap f m = M (\c -> runM m (c . f))

instance Applicative (ConcT n) where
  pure x  = M (\c -> AReturn (c x))
  f <*> v = M (\c -> runM f (\g -> runM v (c . g)))

instance Monad (ConcT n) where
  return  = pure
  m >>= k = M (\c -> runM m (\x -> runM (k x) c))

instance MonadFail (ConcT n) where
  fail e = M (\_ -> AThrow (MonadFailException e))
```

Listing 46: The Déjà Fu continuation monad.

```
pure id <*> v
  = M (\c -> AReturn (c id)) <*> v
  = M (\c -> runM (M (\c -> AReturn (c id))) (\g -> runM v (c . g)))
  = M (\c -> (\c -> AReturn (c id)) (\g -> runM v (c . g)))
  = M (\c -> AReturn ((\g -> runM v (c . g)) id))
  = M (\c -> AReturn (runM v (c . id)))
  = M (\c -> AReturn (runM v c))
 /= v
```

Listing 47: Expansion of the `Applicative` identity law.

5.4. Modelling Concurrent Programs

We represent operations in the `MonadConc` typeclass by an `Action` type, which we give in full in Section 5.5. Each action describes some effect and contains a continuation. A concurrent computation is a sequence of these action values. Each thread is terminated by a distinguished *stop* primitive, which has no continuation. Listing 46 gives the definition and typeclass instances of the Déjà Fu continuation monad[2].

Nonterminating executions Déjà Fu is only able to make scheduling decisions at the level of primitive actions, which means that if evaluating a primitive action does not terminate, Déjà Fu will hang. As Listing 47 shows, we deliberately break the `Applicative` identity law, that `pure id <*> v = v` for all `v`, to make some programs more defined than they otherwise would be.

[2] Déjà Fu aims to support the latest three major releases of GHC, so in the real implementation we use conditional compilation.

```haskell
test = forever (pure "loop") where
  forever mx = mx >> forever mx
```

Listing 48: A simple non-terminating program.

```haskell
newtype Scheduler state = Scheduler
  { scheduleThread
    :: Maybe (ThreadId, ThreadAction)
    -> NonEmpty (ThreadId, Lookahead)
    -> state
    -> (Maybe ThreadId, state)
  }
```

Listing 49: The Déjà Fu `Scheduler` type.

Listing 48 shows a program which is made more defined by breaking the law. Neither >> nor `forever` correspond to primitive actions, so they cannot be pre-empted. If `pure` did not correspond to a primitive action either, then that expression would cause Déjà Fu to loop forever as it tries to compute the continuation. This is an unhelpful result. By breaking the laws and introducing a way to interrupt the `forever` computation, Déjà Fu can instead report that trying to test this program exceeds the execution length limit[3].

Scheduling The choice of which thread to execute is made by a scheduler function. The scheduler is called even if there is only one runnable thread, to keep things simple. A scheduler has the type declared in Listing 49. It is a stateful function which is given the previous action and the runnable threads, which possibly returns a thread to run. If no thread is returned, the computation is aborted. Aborting is used in the implementation of schedule bounding. The state is used in the implementation of DPOR and random scheduling: in the former, the state is a list of scheduling decisions still to make; in the latter, the state is a random number generator.

Success and failure When testing concurrent computations, we are interested in both success and failure. If a computation succeeds and returns a value, we want to know that; if it enters a failure state such as deadlock, we also want to know that. As we have seen earlier, we achieve this by reporting failure: the result of a single execution of the program is a value of type `(Either Failure result, Trace)`.

[3] `https://github.com/barrucadu/dejafu/issues/27` and `issues/113`

5.4.1. Software Transactional Memory

Transactions allow the atomic execution of a sequence of operations involving `TVars`, transactional variables. Unlike operations on `IORefs` or `MVars`, transactions are composable and the whole remains atomic. We express transactions in a similar way to concurrent programs: as a monad of continuations over a primitive action type. As Déjà Fu drives the execution of a concurrent program, it is possible to have arbitrarily complex effects which appear atomic to the program under test.

A transaction evaluates to some success value, an uncaught exception, or an abort. If successful, it may also mutate some transactional variables as a effect; otherwise it does not. If it evaluates to an uncaught exception, we raise the exception in the thread performing the transaction. If it evaluates to an abort, we block the thread performing the transaction until at least one `TVar` read in the transaction is mutated by another thread.

5.4.2. Relaxed Memory

There are three memory models supported in Déjà Fu:

Sequential Consistency
: This model is the most intuitive. A program behaves as a simple interleaving of the actions in different threads. When an `IORef` 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 by the same thread are committed in program order.

Partial Store Order (PSO)
: A relaxation of TSO where each thread has a write buffer for each `IORef`. Writes to different `IORefs` by the same thread may not be committed in program order.

The default memory model for testing is TSO, as that most accurately models the behaviour of modern x86 processors (Owens, Sarkar, and Sewell 2009). The use of a relaxed memory model can require a much larger number of schedules when `IORefs` are shared between threads.

Write buffering We model relaxed memory by introducing buffers for thread writes. When a thread writes to an `IORef`, the write is appended to its buffer. When a thread reads from an `IORef`, it reads the value of the newest write in its buffer, or the most

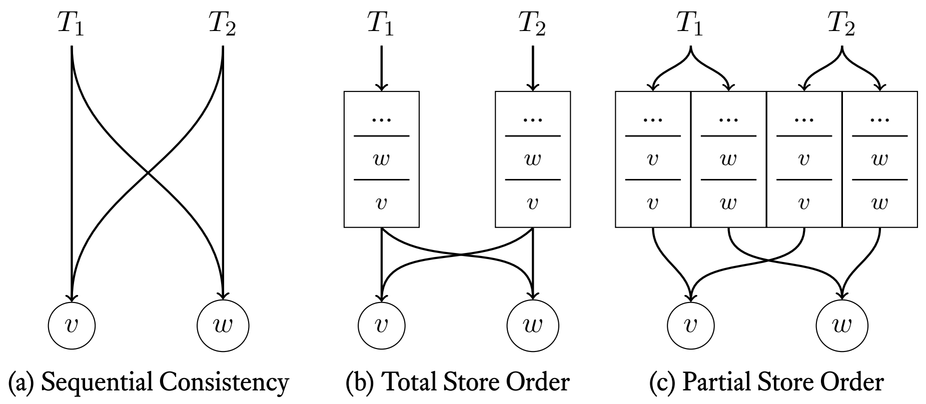


Figure 3: Example of write buffering for two threads and two `IORef`s.

recently committed value if the buffer is empty. After a write is committed, it is removed from its buffer. Any non-empty buffer may have a write committed, but only the oldest write in a buffer may be committed. Figure 3 shows the arrangement of buffers for the three memory models in a system with two threads and two `IORef`s.

We divide operations into three categories: *synchronised* operations impose a *memory barrier*, committing all writes; *partially synchronised* operations commit one or more writes to the same `IORef`; and *unsynchronised* operations never cause a commit.

Phantom threads For each write buffer, we introduce a *phantom thread*. When executed, a phantom thread commits the oldest write from its corresponding buffer. So when using sequentially consistency, the set of runnable threads is exactly the set of threads created by forking which are not blocked, but when using a relaxed memory model, this is not the case. This may seem like an odd approach: why create new threads to model relaxed memory? By using phantom threads, relaxed memory nondeterminism becomes just another aspect of scheduling nondeterminism. We take this approach from Zhang, Kusano, and Wang 2015.

5.5. Operational Semantics

Fundamental to Déjà Fu is an operational semantics for Haskell concurrency, in the form of a step function on primitive actions. Given the current state, which we call the context, we select a thread to execute and either indicate a failure condition, or produce a new context. We also produce an execution trace for the user. Our semantics is similar in

$$
\begin{array}{rcl}
tid,\ tid2 & \in & \text{Thread Identifier} \\
id & \in & \text{Heap Identifier} \\
a & \in & \text{Value} \\
c & \in & \text{Action Value} \\
e & \in & \text{Exception Value} \\
i,\ n & \in & \mathbb{N}_1
\end{array}
$$

$$
\begin{array}{llcl}
\text{Execution Context} & \mathtt{X} & ::= & \langle \mathtt{C},\ \mathtt{B},\ \mathtt{H},\ \mathtt{T} \rangle \\
\text{Capabilities} & \mathtt{C} & ::= & n \\
\text{Buffer (TSO)} & \mathtt{B} & ::= & tid \hookrightarrow [\langle id,\ a \rangle] \\
\text{Buffer (PSO)} & \mathtt{B} & ::= & \langle tid,\ id \rangle \hookrightarrow [a] \\
\text{Heap} & \mathtt{H} & ::= & id \hookrightarrow a \\
\text{Thread Map} & \mathtt{T} & ::= & tid \hookrightarrow \langle \mathtt{K},\ \mathtt{E},\ \mathtt{M} \rangle \\
 & & & \\
\text{Actions} & \mathtt{K} & ::= & c \\
\text{Exception Handlers} & \mathtt{E} & ::= & [\lambda e \to c] \\
\text{Exception Mask} & \mathtt{M} & ::= & \text{Unmasked} \quad | \quad \text{Interruptible} \quad | \quad \text{Uninterruptible}
\end{array}
$$

Figure 4: The syntax of values and execution contexts. Listing 50 gives the definition of action values.

spirit to Vollmer et al. 2017, however we model a much larger set of operations, and support partial store order as well as total store order.

Syntax We express our semantics as transitions on execution contexts. Figure 4 gives the syntax for these contexts. A context has a number of capabilities (`C`), a relaxed-memory buffer (`B`), a heap (`H`), and a collection of threads (`T`). The representation of `B` depends on the memory model: it is a finite mapping, either from thread identifiers to a list of heap identifier and value pairs, or from thread and heap identifier pairs to a list of values; see Section 5.5.2. `H` is a finite mapping from identifiers to values. `T` is a finite mapping from identifiers to threads. A thread has a next operation to perform (`K`), a list of exception handlers (`E`), and a masking state (`M`).

For some rules we include side-conditions or local definitions. We use Haskell syntax for these.

Primitive actions Listing 50 gives the primitive actions for concurrency. Execution happens in the context of an underlying monad, the `n` parameter to the `Action` type.

```
data Action n
  -- Basic multithreading
  = AFork String ((forall b. ConcT n b -> ConcT n b) -> Action n)
      (ThreadId -> Action n)
  | AGetNumCapabilities     (Int -> Action n)
  | ASetNumCapabilities Int        (Action n)
  | AMyTId (ThreadId -> Action n)
  | ALift  (n a)  (a -> Action n)
  | AYield             (Action n)
  | ADelay             (Action n)
  | AReturn            (Action n)
  | AStop

  -- IORefs and relaxed memory
  | forall a.   ANewIORef     String           a (ModelIORef n a -> Action n)
  | forall a.   AReadIORef    (ModelIORef n a)                (a -> Action n)
  | forall a.   AReadIORefCas (ModelIORef n a)    (ModelTicket a -> Action n)
  | forall a.   AWriteIORef   (ModelIORef n a) a                   (Action n)
  | forall a b. AModIORef     (ModelIORef n a) (a -> (a, b))  (b -> Action n)
  | forall a b. AModIORefCas  (ModelIORef n a) (a -> (a, b))  (b -> Action n)
  | forall a.   ACasIORef (ModelIORef n a) (ModelTicket a) a
      ((Bool, ModelTicket a) -> Action n)
  | ACommit ThreadId IORefId

  -- MVars
  | forall a. ANewMVar     String             (ModelMVar n a -> Action n)
  | forall a. APutMVar     (ModelMVar n a) a                  (Action n)
  | forall a. ATakeMVar    (ModelMVar n a)               (a -> Action n)
  | forall a. AReadMVar    (ModelMVar n a)               (a -> Action n)
  | forall a. ATryPutMVar  (ModelMVar n a) a          (Bool -> Action n)
  | forall a. ATryTakeMVar (ModelMVar n a)         (Maybe a -> Action n)
  | forall a. ATryReadMVar (ModelMVar n a)         (Maybe a -> Action n)

  -- Exceptions
  | forall e.   Exception e => AThrow e
  | forall e.   Exception e => AThrowTo ThreadId e (Action n)
  | forall a e. Exception e => ACatching (e -> ConcT n a) (ConcT n a)
      (a -> Action n)
  | forall a.   AMasking MaskingState
      ((forall b. ConcT n b -> ConcT n b) -> ConcT n a) (a -> Action n)
  | AResetMask   MaskingState (Action n)
  | APopCatching               (Action n)

  -- STM
  | forall a. AAtom (ModelSTM n a) (a -> Action n)
```

Listing 50: The primitive concurrency actions.

$$\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{AFork\ \_\ act\ c,E,m0}\rangle]}\rangle \rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c\ tid2,E,m0}\rangle;\ \mathtt{tid2}\mapsto\mathtt{new}]}\rangle$$

where `tid2` is not in the domain of `T`

$\quad$ `reset m'` $=$ `\k -> AResetMask m' k`

$\quad$ `umask mb` $=$ `reset Unmasked >> mb >> \b -> reset m0 >> pure b`

$\quad$ `new` $= \langle$`act umask, [], m0`$\rangle$

$$\begin{aligned}
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{AGetNumCapabilities\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c\ C,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{ASetNumCapabilities\ i\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{i},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{AMyTId\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c\ tid,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{AYield\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{ADelay\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{AReturn\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{c,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{ALift\ na\ c,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{na\ >>=\ c,E,M}\rangle]}\rangle\\
\langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\langle \mathtt{AStop,E,M}\rangle]}\rangle &\rightarrow \langle \mathtt{C},\mathtt{B},\mathtt{H},\mathtt{T}_{[\mathtt{tid}\mapsto\varnothing]}\rangle
\end{aligned}$$

Figure 5: Transition semantics of basic multithreading actions.

This monad is usually `IO`, but can be any `MonadConc`. In these semantics we only use this monad for the `ALift` action. Déjà Fu communicates the result of the computation back to the caller by inserting an `ALift` action which writes the result of the main thread to a mutable reference just before termination.

Our implementation of model execution is more similar to cooperative multitasking than pre-emptive multitasking: if evaluating a primitive action fails to terminate, the entire computation locks up.

Simplifications and omissions The semantic rules we present are for the small-step behaviour. They are tied together by a scheduling loop which we do not present here. This scheduling loop picks a thread to step, steps it, and continues until every thread is blocked or the main thread terminates.

For presentation purposes, we omit some of the complexities of the implementation. We omit building the execution trace, and we omit the definition of helper functions, but explain them when first used. Furthermore, while we present the heap here as a heterogeneous map, in practice we implement it using real mutable references.

5.5.1. Basic Concurrency

Figure 5 shows the semantics for the basic multithreading operations. These rules do not involve memory, so they leave the relaxed-memory buffer and heap untouched. We write $\mathtt{M}_{[\mathtt{k}\mapsto\mathtt{v}]}$ to denote a new map which maps the key `k` to the value `v`.

Rather than have one primitive action for each of the `fork`-like functions, we only provide `AFork`, which implements `forkWithUnmask` for named threads, and implement the others in terms of this. We do not model which capability a thread is executing on, so we do not also need an "`AForkOn`" action.

The `ALift` and `AStop` actions are a little unlike the others. `ALift` causes a Haskell-level effect to occur by executing some action in the underlying monad. `AStop` has no continuation, we use $\mathtt{T}_{[\mathtt{tid} \mapsto \varnothing]}$ to mean that the thread is deleted.

`IORef` operations Figure 6 shows the semantics for the `IORef` operations. We represent a `IORef` as reference to a pair of a number of commits and a latest value. We present the rules for writing to and committing to an `IORef`, which depends on the memory model, in Section 5.5.2

We use $\mathtt{H} \oplus \mathtt{B}$ to mean a new heap with all buffered writes committed, nondeterministically, in some order permitted by the memory model. This is a memory barrier. We use $\mathtt{H} \oplus \mathtt{B}_{[\mathtt{tid}]}$ to mean that only the buffered writes from thread `tid` are committed. Under sequential consistency there are no buffered writes, so $\mathtt{H} \oplus \_ = \mathtt{H}$.

`MVar` operations Figure 7 shows the semantics for the `MVar` operations. We represent an `MVar` as a reference to a `Maybe` value. These operations enforce a memory barrier, and some may cause the thread to block. An action can enforce a memory barrier even if it blocks. We model blocking by not changing the continuation for the chosen thread.

Exceptions and masking Figure 8 shows the semantics for the exception and masking operations. The `raise` function pops from a stack of exception handlers until it finds a handler for the given exception. The `interruptible` function checks if the given thread can be interrupted with an exception. A thread can be interrupted if its masking state is "unmasked", or if its masking state is "interruptible" and it is blocked.

The `AMasking` action is similar to `AFork`: it runs an action after passing an argument to change the masking state. However, unlike `AFork`, `AMasking` runs the action in the current thread rather than creating a new one.

5.5.2. The Memory Model

To avoid giving all of our semantic rules which read from memory three times, we have used the $\oplus$ operator, to denote synchronising any buffered writes to the global memory in a nondeterministic fashion. The behaviour of this operator depends on the memory model used.

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{ANewIORef \_ a c,E,M}\rangle]}\right\rangle \rightarrow \left\langle \texttt{C}, \texttt{B}, \texttt{H}_{[\texttt{id}\mapsto\texttt{(0, a)}]}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c id,E,M}\rangle]}\right\rangle$$
where `id` is not in the domain of `H`

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{AReadIORef id c,E,M}\rangle]}\right\rangle \rightarrow \left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c x,E,M}\rangle]}\right\rangle$$
where $\texttt{(\_, x)} = (\texttt{H} \oplus \texttt{B}_{[\texttt{tid}]})_{[\texttt{id}]}$

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{AReadIORefCAS id c,E,M}\rangle]}\right\rangle \rightarrow \left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c (Ticket id n x),E,M}\rangle]}\right\rangle$$
where $\texttt{(n, x)} = (\texttt{H} \oplus \texttt{B}_{[\texttt{tid}]})_{[\texttt{id}]}$

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{AModIORef id f c,E,M}\rangle]}\right\rangle \rightarrow$$
$$\left\langle \texttt{C}, \varnothing, (\texttt{H} \oplus \texttt{B})_{[\texttt{id}\mapsto\texttt{(n+1, f a)}]}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c,E,M}\rangle]}\right\rangle$$
where $\texttt{(n, a)} = \texttt{H} \oplus \texttt{B}_{[\texttt{tid}]}$

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{AModIORefCAS id f c,E,M}\rangle]}\right\rangle \rightarrow$$
$$\left\langle \texttt{C}, \varnothing, (\texttt{H} \oplus \texttt{B})_{[\texttt{id}\mapsto\texttt{(n+1, f a)}]}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c,E,M}\rangle]}\right\rangle$$
where $\texttt{(n, a)} = (\texttt{H} \oplus \texttt{B})_{[\texttt{id}]}$

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{ACasIORef id (Ticket id n \_) a,E,M}\rangle]}\right\rangle \rightarrow$$
$$\left\langle \texttt{C}, \varnothing, (\texttt{H} \oplus \texttt{B})_{[\texttt{id}\mapsto\texttt{(n+1, a)}]}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c (True, Ticket id (n+1) a),E,M}\rangle]}\right\rangle$$
if $\texttt{n} = \texttt{n0}$
where $\texttt{(n0, \_)} = (\texttt{H} \oplus \texttt{B})_{[\texttt{id}]}$

$$\left\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{ACasIORef id (Ticket id n \_) \_,E,M}\rangle]}\right\rangle \rightarrow$$
$$\left\langle \texttt{C}, \varnothing, \texttt{H} \oplus \texttt{B}, \texttt{T}_{[\texttt{tid}\mapsto\langle\texttt{c (False, Ticket id n0 a0),E,M}\rangle]}\right\rangle$$
if $\texttt{n} \neq \texttt{n0}$
where $\texttt{(n0, a0)} = (\texttt{H} \oplus \texttt{B})_{[\texttt{id}]}$

Figure 6: Transition semantics of `IORef` actions.

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ANewMVar\ \_\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ id,E,M} \rangle]} \right\rangle$$

where `id` is not in the domain of `H`

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ \_}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{APutMVar\ id\ \_\ \_,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{APutMVar\ id\ a\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ a}]} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c,E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ x}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATakeMVar\ id\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ x,E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATakeMVar\ id\ \_\ \_,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ x}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AReadMVar\ id\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ x,E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AReadMVar\ id\ \_\ \_,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ \_}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATryPutMVar\ id\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ False,E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATryPutMVar\ id\ a\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ a}]} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ True,E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Just\ x}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATryTakeMVar\ id\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ (Just\ x),E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{Nothing}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATryTakeMVar\ id\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ Nothing,E,M} \rangle]} \right\rangle$$

$$\left\langle \mathtt{C}, \mathtt{B}, \mathtt{H}_{[\mathtt{id} \mapsto \mathtt{x}]}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ATryReadMVar\ id\ c,E,M} \rangle]} \right\rangle \rightarrow \left\langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c\ x,E,M} \rangle]} \right\rangle$$

Figure 7: Transition semantics of `MVar` actions.

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AThrow\ e}, \mathtt{hs}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \varnothing]} \rangle$$

if `null (raise hs e)`

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AThrow\ e}, \mathtt{hs}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{h\ e}, \mathtt{hs}, \mathtt{M} \rangle]} \rangle$$

if `not (null (raise hs e))`
where `(h:hs)` = `raise hs e`

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AThrowTo\ id\ e\ c}, \mathtt{E}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c}, \mathtt{E}, \mathtt{M} \rangle;\ \mathtt{id} \mapsto \langle \mathtt{AThrow\ e}, \mathtt{E'}, \mathtt{M'} \rangle]} \rangle$$

if `interruptible id`
where $\langle \mathtt{\_}, \mathtt{E'}, \mathtt{M'} \rangle = \mathtt{T}_{[\mathtt{id}]}$

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AThrowTo\ id\ \_\ \_}, \mathtt{E}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \varnothing, \mathtt{H} \oplus \mathtt{B}, \mathtt{T} \rangle$$

if `not (interruptible id)`

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{ACatching\ i\ inner\ c}, \mathtt{hs}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{inner\ (APopCatching\ .\ c)}, \mathtt{h:hs}, \mathtt{M} \rangle]} \rangle$$

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{APopCatching\ c}, \mathtt{\_:hs}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c}, \mathtt{hs}, \mathtt{M} \rangle]} \rangle$$

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AMasking\ m\ act\ c}, \mathtt{E}, \mathtt{m0} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{act\ umask\ (AResetMask\ m0\ .\ c)}, \mathtt{E}, \mathtt{m} \rangle]} \rangle$$

where $\mathtt{m0} = \mathtt{T}_{[\mathtt{tid.M}]}$
`reset m'` = `\k -> AResetMask m' (k)`
`umask mb` = `reset m0 >> mb >> \b -> reset m >> pure b`

$$\langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{AResetMask\ m\ c}, \mathtt{E}, \mathtt{M} \rangle]} \rangle \rightarrow \langle \mathtt{C}, \mathtt{B}, \mathtt{H}, \mathtt{T}_{[\mathtt{tid} \mapsto \langle \mathtt{c}, \mathtt{E}, \mathtt{m} \rangle]} \rangle$$

Figure 8: Transition semantics of exception actions.

$\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle \texttt{AWriteIORef id a c,E,M}\rangle]}\rangle \rightarrow \langle \texttt{C}, \texttt{B}, \texttt{H}_{[\texttt{id}\mapsto\texttt{(0, a)}]}, \texttt{T}_{[\texttt{tid}\mapsto\langle \texttt{c,E,M}\rangle]}\rangle$
if using sequential consistency

$\langle \texttt{C}, \texttt{B}_{[\texttt{tid}\mapsto\texttt{buf}]}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle \texttt{AWriteIORef id a c,E,M}\rangle]}\rangle \rightarrow$
$\langle \texttt{C}, \texttt{B}_{[\texttt{tid}\mapsto\texttt{(buf ++ (id, a))}]}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle \texttt{c,E,M}\rangle]}\rangle$
if using total store order

$\langle \texttt{C}, \texttt{B}_{[\texttt{(tid, id)}\mapsto\texttt{buf}]}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle \texttt{AWriteIORef id a c,E,M}\rangle]}\rangle \rightarrow$
$\langle \texttt{C}, \texttt{B}_{[\texttt{(tid, id)}\mapsto\texttt{(buf ++ a)}]}, \texttt{H}, \texttt{T}_{[\texttt{tid}\mapsto\langle \texttt{c,E,M}\rangle]}\rangle$
if using partial store order

Figure 9: Transition semantics of writing to `IORefs`.

$\langle \texttt{C}, \texttt{B}_{[\texttt{tid}\mapsto\texttt{((id, a):buf)}]}, \texttt{H}_{[\texttt{id}\mapsto\texttt{(n, \_)}]}, \texttt{T}\rangle \rightarrow \langle \texttt{C}, \texttt{B}_{[\texttt{tid}\mapsto\texttt{buf}]}, \texttt{H}_{[\texttt{id}\mapsto\texttt{(n+1, a)}]}, \texttt{T}\rangle$
if using total store order
$\langle \texttt{C}, \texttt{B}_{[\texttt{(tid, id)}\mapsto\texttt{(a:buf)}]}, \texttt{H}_{[\texttt{id}\mapsto\texttt{(n, \_)}]}, \texttt{T}\rangle \rightarrow \langle \texttt{C}, \texttt{B}_{[\texttt{(tid, id)}\mapsto\texttt{buf}]}, \texttt{H}_{[\texttt{id}\mapsto\texttt{(n+1, a)}]}, \texttt{T}\rangle$
if using partial store order

Figure 10: Transition semantics of committing `IORef` writes.

- Under sequential consistency, there are no buffered writes, so $\texttt{H} \oplus \_ = \texttt{H}$.
- Under total store order, we give each thread a buffer, so `B` is a map from thread identifiers to sequences of writes. Here $\texttt{H} \oplus \texttt{B}$ selects a thread with buffered writes nondeterministically, and pops and performs the oldest write from its buffer. This process is repeated until there are no remaining buffered writes.
- Under partial store order, there is a buffer for each (thread, `IORef`) pair, but the process is otherwise the same as for TSO.

Writing to an `IORef` Figure 9 gives the semantic rules for writing to an `IORef`. There is one rule for each memory model, corresponding to the buffering strategy used. Under sequential consistency, the write goes straight to the heap; under TSO, the write is appended to the buffer for the chosen thread; and under PSO the write is appended to the buffer for the chosen (thread, `IORef`) pair.

Committing buffered writes Figure 10 gives the semantic rules for committing buffered `IORef` writes. There is no rule for sequential consistency, as there are no buffered writes in that case. Another way to think of the $\oplus$ operator is that it performs all possible commit actions.

5.5.3. Software Transactional Memory

Here we only give the semantics of the `AAtom` action, a full operational semantics of STM is given in Harris et al. 2005. The STM semantics are small-step reductions of the form $M; \Theta, \Delta \Rightarrow N; \Theta', \Delta'$, where M and N are Haskell expressions, Θ is the heap, and Δ is the set of `TVar` variables which have been accessed. We use $\overset{*}{\Rightarrow}$ to denote a sequence of multiple small-step reductions.

$$\frac{\texttt{stm};\ \texttt{H} \oplus \texttt{B}, \varnothing \overset{*}{\Rightarrow} \texttt{return x};\ \texttt{H'}, \_}{\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid} \mapsto \langle \texttt{AAtom stm c,E,M} \rangle]} \rangle \rightarrow \langle \texttt{C}, \varnothing, \texttt{H'}, \texttt{T}_{[\texttt{tid} \mapsto \langle \texttt{c x,E,M} \rangle]} \rangle}$$

$$\frac{\texttt{stm};\ \texttt{H} \oplus \texttt{B}, \varnothing \overset{*}{\Rightarrow} \texttt{throw x};\ \texttt{H'}, \_}{\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid} \mapsto \langle \texttt{AAtom stm \_,E,M} \rangle]} \rangle \rightarrow \langle \texttt{C}, \varnothing, \texttt{H'}, \texttt{T}_{[\texttt{tid} \mapsto \langle \texttt{AThrow x,E,M} \rangle]} \rangle}$$

$$\frac{\texttt{stm};\ \texttt{H} \oplus \texttt{B}, \varnothing \overset{*}{\Rightarrow} \texttt{retry};\ \_, \_}{\langle \texttt{C}, \texttt{B}, \texttt{H}, \texttt{T}_{[\texttt{tid} \mapsto \langle \texttt{AAtom stm \_,E,M} \rangle]} \rangle \rightarrow \langle \texttt{C}, \varnothing, \texttt{H} \oplus \texttt{B}, \texttt{T} \rangle}$$

Figure 11: Transition semantics for `AAtom`.

Figure 11 gives the semantics of executing transactions. Executing an STM transaction enforces a memory barrier, and writes inside the transaction are synchronised. If the transaction reduces to a `throw` expression, the exception is re-thrown in the thread. As in the `MVar` and exception semantics, a transaction which blocks still imposes a memory barrier.

5.6. Testing Concurrent Programs

Déjà Fu uses a combination of DPOR and schedule bounding to test programs by default. Controlled random scheduling using a fixed number of executions is also available. The testing algorithm to use, and its configuration, is controlled with the `Settings` value (§5.1).

Dependency relation DPOR uses a dependency relation between pairs of actions. Two actions are dependent if the order in which they are performed matters. This relation may have false positives, but cannot have false negatives. False positives lead to exploring redundant executions, false negatives lead to missing distinct ones.

atomicModifyIORef r _	↮	x	if x uses r
atomically x	↮	atomically y	if x writes to a TVar which y accesses
casIORef r _	↮	x	if x uses r
commit _	↮	b	if b enforces a memory barrier
commit r	↮	writeIORef r	if r has no buffered writes
commit r	↮	x	if x uses r and is not a writeIORef r
crefRead r	↮	b	if b enforces a memory barrier and r has buffered writes
crefRead r	↮	x	if x uses r and is not a crefRead r
liftIO _	↮	liftIO _	
modifyIORefCAS r _	↮	x	if x uses r
mvarRead v	↮	mvarRead v	if v is full
mvarRead v	↮	mvarWrite v	
mvarWrite v	↮	mvarWrite v	if v is empty
setNumCapabilities _	↮	getNumCapabilities	
setNumCapabilities _	↮	setNumCapabilities _	
throwTo tgt	↮	x	if x is on thread tgt and can be interrupted
writeIORef r _	↮	x	if x uses r and is not a commit r
x	↮	y	if y ↮ x

Figure 12: The Déjà Fu dependency relation. A `commit` commits one buffered write to a `IORef`. A `crefRead` is a `readIORef` or a `readForCAS`. An `mvarWrite` is a `putMVar` or a `tryPutMVar`. An `mvarRead` is a `takeMVar`, `tryTakeMVar`, `readMVar`, or `tryReadMVar`.

For ease of explanation, DPOR algorithms in the literature are presented for small languages. A paper will typically start with a sentence like "we assume a core concurrent language of reads and writes to shared variables, and locks." For example, in Coons, Madan Musuvathi, and McKinley 2013 two actions are said to be dependent if they are actions of the same thread, or they are both actions on the same shared variable and at least one is a write. The Haskell concurrency API is richer than this, and subtle dependencies (such as which actions impose a memory barrier) are not documented.

Figure 12 shows the dependency relation we use in Déjà Fu. We use a conditional dependency relation (Godefroid and Pirottin 1993). Whether two actions are dependent depends on an approximation of the current state: we record which `IORef`s have buffered writes, whether each `MVar` is full or empty, and what the masking state of every thread is. A conditional dependency relation allows more precise decisions.

Sleep sets The sleep set optimisation (Godefroid 1996) is a complementary approach to DPOR which we use to further reduce schedules explored. We were able to re-use this algorithm without difficulty. The intuition is as follows: if we are in a state s_0 and have a choice of two scheduling decisions, t_1 and t_2, after trying out t_1 there is no point in making the sequence of decisions $t_2 t_1$ from s_0, unless $t_1 \not\leftrightarrow t_2$. All states reachable

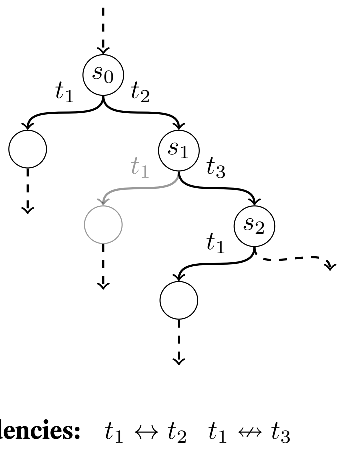

s_0
t_1
t_2
s_1
t_1
t_3
s_2
t_1
Dependencies: $t_1 \leftrightarrow t_2$ $t_1 \not\leftrightarrow t_3$

Figure 13: The sleep set optimisation. Transition t_1 may be pruned in state s_1 but not in state s_2. The transition has been explored from state s_0 and there is no dependent transition between states s_0 and s_1, but there is between states s_0 and s_2. Adapted from Coons, Madan Musuvathi, and McKinley 2013.

from t_1 have already been explored, so the only way a new state could arise is if t_1 had a different effect, which will only be the case if a dependent transition has been taken. Figure 13 shows this graphically.

Formally, we augment each state s with a *sleep set*, containing transitions enabled in s but which we will not make. The initial state has an empty sleep set. Let T be the transitions that have been selected to be explored from s. We proceed as follows: take a transition t_1 out of T. The sleep set associated with the state reached after executing t_1 from s is the sleep set associated with s, minus all transitions that are dependent with t_1. Let t_2 be a second transition taken out of T. The sleep set associated with the state reached after executing t_2 from s is the sleep set associated with s augmented with t_1, minus all transitions that are dependent with t_2. We continue until all transitions in T have been explored, at each step adding the previously taken transitions to the sleep set of the new state, and removing the dependent transitions.

Blocking avoidance Déjà Fu avoids trying schedules which would immediately cause a thread to block without updating any shared state. As a context switch will occur immediately after the thread blocks, there is no point in scheduling it. This is the same insight as in Concuerror (Christakis, Gotovos, and Sagonas 2013), but was discovered independently.

```
main = do
  v <- newEmptyMVar
  fork (myThreadId >> putMVar v "hello world")
  tryReadMVar v
```

Listing 51: A program with a race condition.

```
main = do
  v <- newEmptyMVar
  fork (myThreadId >> myThreadId >> putMVar v "hello world")
  tryReadMVar v
```

Listing 52: Another program with a race condition.

Schedule bounding Déjà Fu supports pre-emption bounding (Madanlal Musuvathi and Qadeer 2007); fair bounding (Madanlal Musuvathi and Qadeer 2008); and depth, or length, bounding (Russell and Norvig 2002). We use a variant of the bounded partial-order reduction algorithm (BPOR) (Coons, Madan Musuvathi, and McKinley 2013), augmented with support for relaxed memory (Zhang, Kusano, and Wang 2015), as our core testing algorithm. All three bounds are enabled by default.

Daemon threads A daemon thread is a thread which is automatically killed after the last non-daemon thread terminates. In Haskell, every thread other than the main thread is a daemon thread, so as soon as the main thread terminates the whole program terminates. This is a problem for DPOR, as it makes the last action of the execution dependent with everything else in the program!

Listing 51 gives a small concurrent program with two possible results: `Nothing`, and `Just "hello world"`. If the scheduler favours the main thread we only see the `Nothing` case, as there is no dependency between `myThreadId` and `tryReadMVar`. Introducing a dependency between the last action of the execution (the `tryReadMVar v` in this case) and everything else solves this problem.

However, these new dependencies also present a difficulty. In Listing 52, the forked thread now performs `myThreadId` twice. If the `tryReadMVar` happens after the second of these, we get the same result as if it happens after the first. Normally, DPOR would recognise this and prune the redundant decision, as `myThreadId` and `tryReadMVar` are independent. However, by introducing a dependency between the final action and everything else, we have said that it is *not* redundant! In general, introducing a dependency like this will lead to many redundant executions which we would otherwise avoid.

The solution we adopt is to change the scheduler. If the scheduler has a choice of

```
S0-----P1--P0-P1-S0-P2--C-S0---P2-P3-P2--S3-P0-P3-P0---S3-P0-P3-S0-
S0----P1-P0-----P2--P0--P2-P0-S2--S3-P1-P0---S1-S3----S0--
S0--------P2-P0--P3-P2-P0-P3-P2-C-S0-S3---S2--S1-C-S1-P0----
```

(a) Original

```
S0----------P1---S2-----S0----S3-----S0--
S0----------P1-P2-----S0--S1--S0---S3-----S0--
S0----------P2--P3-----S0--S2---S1--P0----
```

(b) Simplified

Listing 53: Three execution traces produced by random scheduling and their simplified counterparts.

actions, where one or more will cause the main thread to terminate, it records each decision as a backtracking point. By ensuring that every decision is tried at least once, we do not need to introduce an additional dependency between the final action of the main thread and everything else, and can just let DPOR do its job as usual.

By executing every thread to completion in at least one execution, we do see some redundant schedules. For example, if a thread does not communicate with any other thread, then an execution which schedules it could in principle be modified to not schedule the thread, or perhaps even pruned entirely.

5.7. Execution Traces

Execution traces are not the easiest of things to read, especially if there are many context switches. Traces generated by random scheduling are particularly difficult to read, which is unfortunate, as random testing with a fixed number of executions can be effective for finding bugs and is much faster than DPOR.

Déjà Fu contains a trace simplifier, which attempts to rewrite traces into a shorter form. Unlike the shrinking done in tools like QuickCheck (Claessen and Hughes 2000), Déjà Fu's trace simplification is *semantics-preserving*. An execution trace can be rewritten into a simpler form which is guaranteed to produce the same result, and there is no need to run the program again to verify this. We achieve this by only re-ordering independent actions, using the dependency relation.

Listing 53 shows three execution traces generated by random scheduling, from one of the tests in the Déjà Fu testsuite. These traces are displayed in a pretty-printed form which gives just the essence of the scheduling decisions:

- `Sn`: indicates that thread `n` started executing after the previously executing thread blocked or terminated.
- `Pn`: indicates that thread `n` started executing by pre-empting the previously executing thread.
- `pn`: indicates that thread `n` started executing after the previously executing thread yielded or delayed.
- `C`: indicates the execution of a relaxed memory commit.
- `-`: indicates the execution of one primitive action.

The three original traces involve many context switches and relaxed memory commit actions. They are long and hard to follow. In contrast, the three simplified traces are much shorter, involve far fewer context switches, and have no relaxed memory commit actions at all. The simplified traces are much more useful for program comprehension.

Our trace simplification algorithm has four steps:

1. Rewrite the trace into lexicographic normal form.
2. Prune redundant relaxed memory commit actions.
3. Reduce context switching by pulling actions backwards.
4. Reduce context switching by pushing actions forwards.

We repeat steps (3) and (4) until a fixed point is reached, or a fixed number of iterations has elapsed. Listing 54 gives the full algorithm in pseudocode.

Rewrite the trace into lexicographic normal form The problem of sorting sequences where only some items can be permuted is studied in Anisimov and Knuth 1979. For the case of execution traces, this corresponds to sorting by thread identifier, only re-ordering independent actions. We implement this in much the same way as bubble sort.

Recall that a Haskell program terminates when the main thread does. So by moving actions of the main thread, thread 0, towards the beginning we can then delete a suffix of the trace. This is the main way in which we can produce shorter traces.

Prune redundant relaxed memory commit actions If a commit action is followed by a memory barrier, possibly with some independent actions in between, the commit can be removed if there are no other buffered writes for that `IORef`. The barrier will commit the single write anyway.

Reduce context switching by pulling actions backwards We now walk through the trace from beginning to end, looking for opportunities to pull actions backwards. If we have the trace `[(t1, X), (t2, Y), (t1, Z)]`, where `Y` and `Z` are independent, this transformation re-orders the trace to `[(t1, X), (t1, Z), (t2, Y)]`. We allow any number of independent actions before the action being pulled backwards.

Reduce context switching by pushing actions forwards This is much like the "pull back" transformation. We walk the trace, looking for cases where actions can be pushed forward. If we have the trace `[(t1, X), (t2, Y), (t1, Z)]`, where `X` and `Y` are independent, this transformation re-orders the trace to `[(t2, Y), (t1, X), (t1, Z)]`.

Having both the "pull back" and "push forward" transformations is not redundant. If we have three actions—`X`, `Y`, and `Z`—it may be that `X` is independent with `Y` but `Z` is not, or vice versa.

5.8. Soundness and Completeness

Correctness for Déjà Fu comes down to two core issues: (1) is the set of results which are possible under normal execution the same as the set of results which are possible under Déjà Fu; and (2) will Déjà Fu discover all these possible results? These questions highlight a natural separation in the theory: the relation between Déjà Fu's concurrency semantics and GHC's, and the correctness of the machinery to discover new executions.

We do not attempt to formally verify the Déjà Fu implementation, but we do have an extensive test suite. Most of our tests consist of nondeterministic programs, where we verify that Déjà Fu finds precisely the behaviours we expect.

Property tests Déjà Fu is not one monolithic implementation. It has components which can be specified and tested in isolation. For example, the dependency relation must be commutative. We use Hedgehog (Stanley 2018), a randomised property-testing library, to check these properties of the implementation.

Integration tests Most of the tests are integration tests, consisting of a small concurrent program and a property to check. A failure in such a test could in principle be anywhere in Déjà Fu, but in practice failures in different components tend to manifest differently. A failure in the DPOR implementation tends to manifest as invalid schedules being generated; whereas a failure in the concurrency implementation tends to manifest as incorrect results of individual executions.

```
function SIMPLIFY($T$)
    # rewrite in lexicographic normal form
    $r \leftarrow$ true
    while $r$ do
        $r \leftarrow$ false
        for $i \leftarrow 0 \ldots \textsc{Length}(T) - 2$ do
            if $\textsc{tid}(T_i) > \textsc{tid}(T_{i+1}) \wedge T_i \leftrightarrow T_{i+1}$ then
                $\textsc{Swap}(T, i, i+1)$
                $r \leftarrow$ true
    # prune redundant commits
    $i \leftarrow 0$
    while $i < \textsc{Length}(T) - 1$ do
        if $\textsc{IsCommit}(T_i)$ then
            if $T_i \leftrightarrow T_{i+1}$ then
                $\textsc{Swap}(T, i, i+1)$
            else if $\textsc{IsBarrier}(T_{i+1}) \wedge \textsc{ioref}(T_i)$ has only this buffered write then
                $\textsc{Delete}(T, i)$
                CONTINUE
        $i \leftarrow i + 1$
    # reduce context switching
    $r \leftarrow$ true
    while $r \wedge \neg$ iteration limit is reached do
        $r \leftarrow$ false
        # pull actions backwards
        for $i \leftarrow 0 \ldots \textsc{Length}(T) - 2$ do
            if $\textsc{tid}(T_i) \neq \textsc{tid}(T_{i+1})$ then
                if $j \leftarrow \textsc{Find}(T, i+1, \lambda j.\textsc{tid}(T_i) = \textsc{tid}(T_j))$ then
                    $\textsc{Move}(T, j, i+1)$
                    $r \leftarrow$ true
        # push actions forwards
        $i \leftarrow 0$
        while $i < \textsc{Length}(T) - 1$ do
            if $\textsc{tid}(T_i) \neq \textsc{tid}(T_{i+1})$ then
                if $j \leftarrow \textsc{Find}(T, i+1, \lambda j.\textsc{tid}(T_i) = \textsc{tid}(T_j), T_i)$ then
                    $\textsc{Move}(T, i, j-1)$
                    $r \leftarrow$ true
                    CONTINUE
            $i \leftarrow i + 1$
```

Listing 54: The trace simplification algorithm. SWAP swaps two entries by index. MOVE moves an element to a new index, shifting all other elements. DELETE removes an element by index. FIND$(T, i, \lambda \ldots)$ finds the first element, starting from index i, which meets the predicate, and returns its index if it is independent with all the intervening elements. FIND$(T, i, \lambda \ldots, a)$ finds the first element, starting from index i, which meets the predicate, and returns its index if a is independent with all the intervening elements.

Our integration tests fall into the following classes:

- Single threaded tests for the `MonadConc` primitives, ensuring that only the single correct behaviour is observed.
- Multi-threaded tests for the `MonadConc` primitives, ensuring that the expected nondeterminism is observed.
- So-called *litmus tests* for the relaxed memory implementation, ensuring that all expected relaxed behaviours are observed.
- A copy of the async library's (Marlow 2017) test suite, for our `MonadConc` version.
- A collection of refinement tests from CoCo, which we shall discuss in Chapter 7.
- Finally, a collection of regression tests for previous bugs.

Example programs Finally, we have a collection of larger integration tests which serve also as example usages of Déjà Fu, three of which we discuss in Section 5.9: the monad-par library (Marlow and R. R. Newton 2016; Marlow, R. Newton, and Peyton Jones 2011), the auto-update library (Snoyman 2016), and the async library (Marlow 2017).

5.8.1. Correct Execution

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

Program behaviour There is no standard for concurrent Haskell. There is only what GHC provides. The behaviour of many operations is clear, but for some it is not. `IORef` operations are particularly complicated, as their behaviour depends on the underlying memory model, which is unspecified. We chose TSO, and assume that the GHC optimiser or code generator does not affect the memory model.

There are some intentional semantic differences for practical reasons. For example, GHC can sometimes detect deadlocks involving only a subset of the threads. We cannot do this. Although the behaviour of Déjà Fu is not correct with respect to GHC-compiled behaviour in all cases, we claim it is close enough to be useful.

Possible executions Our stepwise execution of concurrent programs allows a scheduling decision to be made between each primitive action, which does not correspond to how GHC handles scheduling:

GHC implements pre-emptive multitasking: the execution of threads are

```
bottom = do
  fork (last [1..])
  pure ()
```

Listing 55: A program that does not halt under Déjà Fu but does under GHC.

> 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). (The GHC Developers 2017a)

So there are executions involving the pre-emption of the evaluation of non-terminating expressions which are possible under GHC but not under Déjà Fu. However, Déjà Fu is even worse than this with bottom values. The program in Listing 55 will fail to terminate, even if the thread with the infinite computation is never scheduled, as Déjà Fu will hang trying to compute the continuation so it can call the scheduler.

5.8.2. Correct Testing

Correctness of testing asks: are the schedule prefixes generated by the DPOR machinery valid? Furthermore, are there any possible results for which no schedule will be generated? This is different to the testing framework generating every schedule, as that is precisely what DPOR tries to avoid.

Prefix validity Executions are stored internally as a stack, shown in Listing 56. The sequence of thread IDs corresponding to this stack represents a complete execution of the program. There is a unique initial state, where only the initial thread is runnable and nothing has been done. We have some well-formedness properties:

1. Every thread in the to-do set is runnable.
2. Every thread in the done set is runnable.
3. The taken set is a subset of the done set.
4. The done and to-do sets are disjoint.
5. The next-taken thread, if there is one, is in the done set.

These properties should hold inductively over the whole state. We check these invariants everywhere a `DPOR` value is constructed, and abort if the state is invalid. In the Déjà Fu testsuite, the execution time overhead of this checking is 3%.

```
data DPOR = DPOR
  { dporRunnable :: Set ThreadId
  -- ^ What threads are runnable at this step.
  , dporTodo     :: Map ThreadId Bool
  -- ^ Follow-on decisions still to make, and whether that decision
  -- was added conservatively due to the bound.
  , dporNext     :: Maybe (ThreadId, DPOR)
  -- ^ The next decision made.
  , dporDone     :: Set ThreadId
  -- ^ All transitions which have been taken from this point,
  -- including conservatively-added ones.
  , dporSleep    :: Map ThreadId ThreadAction
  -- ^ Transitions to ignore until a dependent transition happens.
  , dporTaken    :: Map ThreadId ThreadAction
  -- ^ Transitions which have been taken, excluding
  -- conservatively-added ones.
  } deriving (Eq, Show)
```

Listing 56: The DPOR state is a stack of scheduling decisions.

The prefixes we generate are sequences of taken decisions followed by a single to-do decision. By maximising the length of prefixes, we obtain a depth-first search of the space of schedules. Provided the well-formedness properties hold, and the runnable sets are correctly recorded during execution, then a generated schedule prefix will be valid.

Schedule completeness The DPOR machinery should eventually find every possible result of a given program. However, as schedule bounding is involved, some results may not be reached. So instead we require that, for all sets of bounds, all results possible subject to those bounds show up under testing with the same bounds. Our core testing algorithm satisfies this property (Coons, Madan Musuvathi, and McKinley 2013).

5.9. Case Studies

We now discuss the process and results of applying Déjà Fu to three Haskell libraries:

1. We identify and fix a deadlock in the monad-par library (Marlow and R. R. Newton 2016; Marlow, R. Newton, and Peyton Jones 2011).
2. We reproduce a known deadlock in the auto-update library (Snoyman 2016).
3. We use property-based testing to reproduce a known bug in the async library (Marlow 2017).

We chose these libraries because each is by proficient Haskell programmers well versed with concurrency, and yet they all contain unintentional bugs. This shows that even those familiar with the standard pitfalls of concurrent programming encounter problems.

None of these libraries is written using the `MonadConc` abstraction, so we had to modify the existing code before we could test them with Déjà Fu.

5.9.1. monad-par

The monad-par library (Marlow and R. R. Newton 2016; Marlow, R. Newton, and Peyton Jones 2011) provides a traditional-looking concurrency abstraction, giving the programmer threads and mutable state, however it is deterministic. Determinism is enforced by restricting shared state: it is an error to write more than once to the same mutable variable, and read operations block until a value has been written. Programs written using the library will either give a deterministic result, or terminate with a multiple-write error. These shared variables, called `IVars`, implement futures (Marlow, R. Newton, and Peyton Jones 2011). Despite its limitations, the library can be effective in speeding up pure code (Marlow, R. Newton, and Peyton Jones 2011).

The library provides six different schedulers. We ported the "direct" scheduler, a work-stealing scheduler, to the `MonadConc` typeclass. This was a straightforward and compiler-driven refactoring. Changing function types to use `MonadConc` rather than `IO` led to compiler errors showing where the next changes needed to be made. We iterated this process of fixing errors and recompiling until the library successfully compiled once more. Changes were needed in two of the source files.

Some simplifications were made in the conversion process:

- The scheduler creates a pseudorandom number generator for each worker thread. As systematic testing requires that the scheduler be the only source of nondeterminism, we fixed the random seeds: the first worker thread gets the seed zero, the second gets the seed one, and so on.
- The scheduler uses the C pre-processor to choose between different implementations of some of its functionality. There are nine flags, each of which are independent. We only ported and tested the default configuration.
- The scheduler includes some debugging code for detecting and reporting errors. We removed it.

Listing 61 on page 99 shows the original and converted versions of the scheduler initialisation code. As can be seen, they are similar, even though this is a core component of a rather sophisticated library, where the types have been changed. Table 4 breaks down

	Direct.hs	DirectInternal.hs
Language extensions	1	0
Module imports	6	1
Type definitions	7	13
Type signatures	32	7
Function renames	2	0
Logic changes	16	0
Total	75	26

Table 4: Breakdown of changes to port the monad-par "direct" scheduler.

```
parfilter :: (MonadConc m, NFData a) => (a -> Bool) -> [a] -> Par m [a]
parfilter _ []  = pure []
parfilter f [x] = pure (if f x then [x] else [])
parfilter f xs  = do
    let (as, bs) = halve xs
    v1 <- Par.spawn (parfilter f as)
    v2 <- Par.spawn (parfilter f bs)
    left  <- Par.get v1
    right <- Par.get v2
    pure (left ++ right)
  where
    halve xs = splitAt (length xs `div` 2) xs
```

Listing 57: An example usage of the monad-par library.

the changes across both files. Code changes are broken down into "renames," where the concurrency library simply provides a different name for a function, and "logic," where that was not the case. The logic changes were either: (1) places where a call to `liftIO` or `lift` was now necessary; or (2) inserting uses of `getNumCapabilities`.

Listing 57 shows an example usage of the library. This `parfilter` function filters a list in parallel, using a divide-and-conquer approach. If the list is not empty or a singleton, it is split in half and a new thread created to filter each half. The results of each new thread are combined to produce the overall result. The library requires all shared state have an instance of the `NFData` typeclass, which provides an operation to evaluate data to normal form. The library gains its speed by evaluating data in separate threads.

Finding a deadlock Listing 58 shows a test case for `parfilter`. A parallel filter should produce the same result as a normal filter. Table 5 shows performance measurements for our test case, with the input list `[0..5]`, in six different configurations.

```
test_parmonad :: (MonadConc m, MonadIO m) => [Int] -> m Bool
test_parmonad xs = do
    let p x = x `mod` 2 == 0
    s <- runPar (parfilter p xs)
    pure (s == filter p xs)
```

Listing 58: A test case comparing a parallel filter to a normal filter.

	Schedules	Deadlocks	Time (s)	Max Residency (MB)
Bounded DPOR	6140	0	35.1	339
Unbounded DPOR				$\geq$ 16GiB
Swarm	100	2	0.17	21

(a) Keeping all execution traces in memory.

	Schedules	Deadlocks	Time (s)	Max Residency (kB)
Bounded DPOR	6140	0	27.9	842
Unbounded DPOR			$\geq$ 48hrs	
Swarm	100	2	0.14	1600

(b) Only keeping buggy execution traces in memory.

Table 5: Performance of the monad-par case study with three different exploration tactics. Unbounded DPOR was aborted in both cases, after consuming too many resources.

The numbers do not look so good for DPOR. Swarm managed to find two deadlocking executions out of 100, whereas bounded DPOR found none in 6140. Unbounded DPOR did not complete at all: it rapidly consumed all the memory of the host system when keeping traces in memory, and was still running after two days while discarding traces.

When we inspect one of the execution traces leading to deadlock, we gain two clues for why DPOR performs poorly: (1) the trace is 793 entries long, but the length bound for DPOR is 250; and (2) there are 473 calls to `liftIO`, Déjà Fu considers all `IO` actions dependent and so tries every possible interleaving of these.

Following the code by eye as we read a 793-entry trace is not realistic. So instead, let's look at the last few entries in the trace, shown in Listing 59. Each trace entry is a tuple consisting of the scheduling decision made, the alternative decisions possible, and what the thread did. As this is right before deadlock, it is not surprising that there are no other threads which could be scheduled. There are only four calls to `getNumCapabilities` in the code we ported. We can look at each, to find the code which matches the trace.

```
(Continue,[],LiftIO)
(Continue,[],LiftIO)
(Continue,[],LiftIO)
(Continue,[],ReadIORef 2)
(Continue,[],LiftIO)
(Continue,[],NewMVar 7)
(Continue,[],ModIORef 4)
(Continue,[],GetNumCapabilities 2)
(Continue,[],BlockedTakeMVar 7)
```

Listing 59: The final ten entries of the deadlocking monad-par trace.

```haskell
go 0 _ | _IDLING_ON =
  do m <- newEmptyMVar
     r <- modifyHotVar idle (\is -> (m:is, is))
     numCapabilities <- getNumCapabilities
     if length r == numCapabilities - 1
       then do
         mapM_ (\vr -> putMVar vr True) r
       else do
         done <- takeMVar m
         if done
           then do
             return ()
           else do
             i <- getnext (-1::Int)
             go (maxtries numCapabilities) i
```

Listing 60: The source of the deadlock in the monad-par library. In the "then" branch of the conditional, the `idle` list is not emptied when waking every blocked thread.

Listing 60 is our match. When a thread is unable to steal work, it creates an empty `MVar` which it adds to a shared list, called `idle`; if that list already has an entry for every other thread, they are woken by writing a value to their `MVar`; otherwise, the thread blocks by calling `takeMVar` on its new, empty, `MVar`. The final fragment of our execution trace corresponds to this code where `length r == numCapabilities - 1` is false. But how can that be false? *The list is never emptied!* Think about what happens if every thread reaches this logic *twice*:

1. `n - 1` threads add themselves to the list, and block.
2. The final thread adds itself to the list, and wakes up the other threads.
3. **The list is not emptied, even though every thread is woken.**
4. `n - 1` threads add themselves to the list, again, and block.
5. The final thread adds itself to the list. It does *not* trigger the wake-up logic, because the list is not the right length, and so it also blocks.
6. **Every thread is now blocked.**

We can confirm our suspicion by checking the trace. Each thread does exhibit that pattern twice, so our deduction is correct. This problem is solved by writing `[]` to `idle` before waking the threads. The write must happen before, otherwise there is a new race condition: one of the woken threads could add itself to the list again, and then its `MVar` be lost when the list is cleared.

Handling an exception When running our test case in a loop using `IO`, to verify that we really had fixed the problem, another issue arose which Déjà Fu did *not* find. After a while, the main thread would be killed by a `BlockedIndefinitelyOnMVar` exception. Such exceptions are out of scope (§5.1.2), so Déjà Fu could never find this new problem. There is always a gap between the real system and the model. Déjà Fu is just one component of the Haskell programmer's toolbox, it is not the be-all and end-all for concurrency testing. However, for the class of bugs which Déjà Fu can find, it is much more effective than simply running the program in `IO` many times.

By turning on the library's debugging output, and adding some more of our own, we were able to track the problem down to the same logic as before, Listing 60. A thread was still getting blocked here, despite our fix for the deadlock. How can a thread get blocked indefinitely there if we make sure the last thread wakes up the others? *The number of workers is assumed to be equal to the number of capabilities!* If even a single worker terminates, all the others will block.

Fortunately, the relevant source code is not extensive. We were able to quickly ascertain that the library divides its workers into two categories: there is a single main worker, which communicates the result of the computation back to its caller and terminates when done; and there are all the other workers. The other workers do check if the computation is complete, but only in certain places. So this was happening:

1. A non-main worker checks if the computation is complete, and sees that it is not.
2. The same worker blocks itself as usual.
3. The computation finishes, and the main worker terminates.
4. **The GHC runtime delivers an exception to the blocked worker.**

It is harmless for the worker to be blocked at this point, as the overall computation is long-complete, and the result communicated back to the user. However, each worker thread is given an exception handler which throws any received exception to its creator. In this case, the creator was the main thread, so the whole program is terminated. The solution is to check if the computation has terminated before blocking.

5.9.2. auto-update

The auto-update library (Snoyman 2016) runs tasks periodically, but only if needed. For example, a web server may handle each request in a new thread, and log the time that the request arrives. Rather than have every such new thread check the time, one thread could be created to update a single shared `IORef` every second. However, if the request frequency is less than once per second, this is wasted work. The library allows defining a periodic action which only runs when needed.

The implementation, excluding comments and imports, is reproduced in Listing 64. The library defines a function, `mkAutoUpdate`, which forks a worker thread to perform the action when required. The function returns an `IO` action to read the current result, if necessary blocking until there is one. The transformation to the `MonadConc` typeclass is straightforward, and we omit it here.

Listing 62 shows an example usage of the `MonadConc` version of the library. The `defaultUpdateSettings` value describes an auto-updater which runs every second, producing the value `()`. An `MVar` is used to communicate to the thread that the updater should run. Inside the worker, a delay is used to ensure that the action is not computed too frequently: this is what gives the rate limiting. So we create an auto-updater which produces `()`, and immediately demand the value.

```
makeScheds :: Int -> IO [Sched]
makeScheds main = do
   workpools <- replicateM numCapabilities $ R.newQ
   rngs <- replicateM numCapabilities $ Random.create >>= newHotVar
   idle <- newHotVar []
   sessionFinished <- newHotVar False
   sessionStacks   <- mapM newHotVar
     (replicate numCapabilities [Session baseSessionID sessionFinished])
   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
                             sessionCounter, activeSessions
                           }
                   | x   <- [0 .. numCapabilities-1]
                   | wp  <- workpools
                   | rng <- rngs
                   | stck <- sessionStacks
                   ]
   pure allscheds
```

(a) Original

```
makeScheds :: (MonadConc m, MonadIO m) => Int -> m [Sched m]
makeScheds main = do
   numCapabilities <- getNumCapabilities
   workpools <- replicateM numCapabilities $ liftIO R.newQ
   let rng i = liftIO (Random.initialize (V.singleton $ fromIntegral i))
   rngs <- mapM (\i -> rng i >>= newHotVar) [0..numCapabilities]
   idle <- newHotVar []
   sessionFinished <- newHotVar False
   sessionStacks   <- mapM newHotVar
     (replicate numCapabilities [Session baseSessionID sessionFinished])
   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
                             sessionCounter, activeSessions
                           }
                   | x   <- [0 .. numCapabilities-1]
                   | wp  <- workpools
                   | rng <- rngs
                   | stck <- sessionStacks
                   ]
   pure allscheds
```

(b) Déjà Fu

Listing 61: The monad-par "direct" scheduler initialisation.

```haskell
test_autoupdate :: MonadConc m => m ()
test_autoupdate = do
  auto <- mkAutoUpdate defaultUpdateSettings
  auto
```

Listing 62: An example usage of the auto-update library.

```
> autocheck test_autoupdate

[fail] Never Deadlocks
        [deadlock] S0--------S1-----------S0-
[pass] No Exceptions
[fail] Consistent Result
        () S0--------S1--------p0--

        [deadlock] S0--------S1-----------S0-
```

Listing 63: Using Déjà Fu to run a collection of standard tests. The `autocheck` function looks for deadlocks, uncaught exceptions in the main thread, and nondeterminism. Each result is displayed with a simplified view of a representative execution trace.

Testing with Déjà Fu Listing 63 shows one way in which we can use Déjà Fu to explore the behaviour of our small example. The `autocheck` function looks for some common concurrency errors. In this example, Déjà Fu discovers a deadlock. Each result is displayed with a simplified view of a representative execution trace. More detailed execution traces are also available, which contain a summary of the primitive actions which occurred and the alternative scheduling decisions available.

We can see from the trace of the deadlock result that thread 0 executed for a while, then thread 1, then thread 0 again. As these are all `S` points, each thread executed until it blocked. So we can look at the source code in Listing 62 and Listing 64 to see what Following the execution by eye, we see this sequence of concurrency events:

1. Thread 0:
   (a) Line 16: `currRef <- newIORef Nothing`
   (b) Line 17: `needsRunning <- newEmptyMVar`
   (c) Line 18: `lastValue <- newEmptyMVar`
   (d) Line 20: `void $ forkIO $ ...`
   (e) Line 35: `mval <- readIORef currRef`
   (f) Line 39: `void $ tryPutMVar needsRunning ()`
   (g) Line 40: `readMVar lastValue`
   (h) **Thread 0 is now blocked, as `lastValue` is empty.**

2. Thread 1:
    (a) Line 21: `takeMVar needsRunning`
    (b) Line 25: `writeIORef currRef $ Just a`
    (c) Line 26: `void $ tryTakeMVar lastValue`
    (d) Line 27: `putMVar lastValue a`
    (e) **Thread 0 is now unblocked, as `lastValue` is full.**
    (f) Line 29: `threadDelay $ updateFreq us`
    (g) Line 31: `writeIORef currRef Nothing`
    (h) Line 32: `void $ takeMVar lastValue`
    (i) **Thread 0 is still unblocked, even though `lastValue` is now empty again.**
    (j) **Thread 1 now loops.**
    (k) Line 21: `takeMVar needsRunning`
    (l) **Thread 1 is now blocked, as `needsRunning` is empty.**
3. Thread 0:
    (a) Line 40: `readMVar lastValue`
    (b) **Thread 0 is now blocked, as `lastValue` is empty.**

Both threads are blocked, so the computation is deadlocked. The other result shown in Listing 63 occurs if thread 0 starts executing after thread 1 delays. So the root cause of this deadlock is clear: deadlock may occur if the call to `threadDelay` on line 29 completes before the other thread resumes execution. Despite this bug being rather simple, not requiring any pre-emptions at all to trigger, it arose in practice. How easy it is to make mistakes when implementing concurrent programs!

Performance of testing Table 6 shows performance measurements for our test case in six different configurations. Both the library itself and our test case are small, so it is perhaps no surprise to see that in all configurations, execution only takes a fraction of a second. We can see the effect of the schedule bounding: when the bounds are disabled, the number of schedules tried almost doubles, and two new deadlocking executions are found.

To reduce memory usage, Déjà Fu is able to discard results or execution traces which the user considers uninteresting in some way. Table 6b shows the impact of this change, where we have designated non-deadlocking traces as uninteresting. The effect is particularly significant in the Swarm case, suggesting that Swarm may tend to find longer execution traces than DPOR.

```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

    pure $ 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 -> pure $ throw (e :: SomeException)
```

Listing 64: The implementation of the auto-update package.

	Schedules	Deadlocks	Time (s)	Max Residency (kB)
Bounded DPOR	49	18	0.006	119
Unbounded DPOR	80	20	0.008	124
Swarm	100	20	0.008	1297

(a) Keeping all execution traces in memory.

	Schedules	Deadlocks	Time (s)	Max Residency (kB)
Bounded DPOR	49	18	0.006	71
Unbounded DPOR	80	20	0.006	63
Swarm	100	20	0.006	108

(b) Only keeping buggy execution traces in memory.

Table 6: Performance of the auto-update case study with three different exploration tactics.

5.9.3. async

The async library (Marlow 2017) allows programmers to write asynchronous code without needing to worry about details such as threads, shared state, or exceptions. Listing 65 shows a typical usage of the library. The `async` function begins executing an `IO` action in a new thread. The `wait` function blocks until the action is done and returns the result. If the action throws an exception, `wait` also throws the exception. There is a third basic operation: `cancel`, which terminates the thread associated with an asynchronous action.

Using these three building blocks of `async`, `wait`, and `cancel`, the library provides a collection of higher-level abstractions which are widely used in the Haskell ecosystem. One of these higher-level abstractions is the `Concurrently` type, a simple wrap-

```haskell
downloadBoth :: URL -> URL -> IO (String, String)
downloadBoth url1 url2 = do
  a1 <- async (download url1)
  a2 <- async (download url2)
  page1 <- wait a1
  page2 <- wait a2
  pure (page1, page2)
```

Listing 65: A typical usage of the async library. Both URLs are downloaded concurrently in separate threads.

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap f a = do
  f' <- f
  a' <- a
  pure (f' a')
```

Listing 66: The `ap` function.

per around `IO`, which allows concurrently composing actions with the `Applicative` `<*>` operator. The two arguments to `<*>` are computed concurrently in separate threads, and then combined. Listing 71 on page 108 gives the implementation of `Concurrently`.

In Haskell, we like our typeclasses to have *laws* specifying how instances should behave. Without such a specification, it is impossible to write typeclass-polymorphic functions with any reasonable expectation of what will happen. These laws are not checked by the compiler: GHC is no theorem prover. Rather, it is up to the author of a typeclass instance to ensure that they follow all appropriate laws. Failure by a library author to follow the laws can lead to unexpected behaviour for users. As `Applicative` is a superclass of `Monad`, it should come as no surprise that there is a law relating the two: specifically, that `<*> = ap`. The definition of `ap` is given in Listing 66.

The `ap` law does *not* hold for the `Concurrently` monad[4]! With the benefit of hindsight, the cause is clear: `<*>` runs its arguments concurrently, whereas `ap` has no choice but to run its arguments sequentially. If the two arguments can concurrently interfere with each other, then `<*>` exhibits more nondeterminism than `ap`.

Property-testing typeclass laws Could Déjà Fu have helped here? We believe so. By changing the `Concurrently` type to be polymorphic over the underlying monad, we can substitute in any `MonadConc`. We can then test the laws. We used QuickCheck (Claessen and Hughes 2000) for this, but any property-testing tool which can generate functions and check monadic properties would do.

Listing 67 shows a *passing* property. There is no concurrent interference between the two arguments to `<*>` and `ap`, so the bug does not manifest. For the reader unfamiliar with `QuickCheck`: `Fun a b` represents a function of type `a -> b`. The property runs both concurrent actions with Déjà Fu, and compares the sets of results. The property passes if and only if the sets of results are equal. This is one way in which two concurrent programs can be equivalent, we discuss this idea further in Chapter 7 where we extend our notion of equivalence to stateful computations.

[4] https://github.com/simonmar/async/pull/26

```
prop_monad_ap1 :: Ord b => Fun a b -> a -> Property
prop_monad_ap1 (apply -> f) a = (pure f <*> pure a) `eq` (pure f `ap` pure a)

eq :: Ord a => Concurrently ConcIO a -> Concurrently ConcIO a -> Property
eq (Concurrently left) (Concurrently right) = monadicIO $ do
  l <- resultsSet defaultWay defaultMemType left
  r <- resultsSet defaultWay defaultMemType right
  assert (l == r)
```

Listing 67: The `<*>` = `ap` law, with no concurrent interference.

```
prop_monad_ap2 :: Ord b => Fun a b -> Fun a b -> a -> Property
prop_monad_ap2 (apply -> f) (apply -> g) a = go (<*>) `eq` go ap where
  go combine = do
    flagvar <- newEmptyMVar
    let cf = do { flag <- tryPutMVar flagvar (); pure (if flag then f else g) }
    let ca = do { tryPutMVar flagvar (); pure a }
    pure (Concurrently cf `combine` Concurrently ca)
```

Listing 68: The `<*>` = `ap` law, with concurrent interference.

Our `prop_monad_ap1` property is uninteresting in a sense because it is clearly free from concurrency errors: the very errors which we want to detect! It is free of them because the arguments to `<*>` and `ap` are pure values, so there can be no concurrent interference between them. To observe the law being broken, we must create a race condition.

Listing 68 contains a race condition. We now generate two functions with QuickCheck. When executing the concurrent action, we use an `MVar` to decide which function to use. If the `MVar` is empty we use the first function, if it is full we use the second. If the combining function, `<*>` or `ap`, executes its arguments concurrently we will see both functions tried; if it executes its arguments sequentially, we will only see the first function. Indeed, we do see the bug. Listing 69 gives the QuickCheck and Déjà Fu outputs.

Performance of testing Table 7 shows performance measurements for a variant of our test case. We cannot give the property itself to Déjà Fu, so we extract the `MonadConc` computation and hard-code the parameters generated by QuickCheck, shown in Listing 70. As in the auto-update case study, this is a small test case, so it is perhaps unsurprising that it is fast and requires little memory. Once again, we see that Swarm requires more memory than DPOR, even taking the increased number of schedules tried into account.

```
> quickCheck prop_monad_ap2
*** Failed! Falsifiable (after 3 tests and 8 shrinks):
{_->""}
{_->"a"}
0
```

(a) The QuickCheck output.

```
> resultSet defaultWay defaultMemType (go (<*>) (\_ -> "") (\_ -> "a") 0)
fromList [Right "",Right "a"]

> resultSet defaultWay defaultMemType (go ap (\_ -> "") (\_ -> "a") 0)
fromList [Right "a"]
```

(b) The Déjà Fu output.

Listing 69: The result of the failing `<*> = ap` property.

The benefit of hindsight We have the benefit of knowing about the bug, leading us to the correct test. Is it unrealistic to expect a user to have the foresight to write a test like this in the beginning? We think not. When implementing functions for combining concurrent actions, it is no great leap to wonder what happens if there are races between these actions. The property may appear contrived, but it is a natural way to investigate the effect of a race condition in the `<*> = ap` law.

	Schedules	Failures	Time (s)	Max Residency (kB)
Bounded DPOR	17	2	0.002	131
Unbounded DPOR	30	3	0.004	135
Swarm	100	37	0.012	1360

(a) Keeping all execution traces in memory.

	Schedules	Failures	Time (s)	Max Residency (kB)
Bounded DPOR	17	2	0.004	92
Unbounded DPOR	30	3	0.009	85
Swarm	100	37	0.011	648

(b) Only keeping buggy execution traces in memory.

Table 7: Performance of the `<*> = ap` test case with three different exploration tactics.

```
test_concurrently :: MonadConc m => m Bool
test_concurrently = do
    l <- go (<*>)
    r <- go ap
    pure (l == r)
  where
    go combine = runConcurrently $ do
      flagvar <- newEmptyMVar
      let cf = do { flag <- tryPutMVar flagvar (); pure (if flag then f else g) }
      let ca = do { tryPutMVar flagvar (); pure a }
      pure (Concurrently cf `combine` Concurrently ca)

    f = \_ -> ""
    g = \_ -> "a"
    a = 0
```

Listing 70: The <*> = ap test case, with the generated functions hard-coded.

```haskell
newtype Concurrently a = Concurrently { runConcurrently :: IO a }

instance Functor Concurrently where
  fmap f (Concurrently a) = Concurrently (fmap f a)

instance Applicative Concurrently where
  pure = Concurrently . pure

  Concurrently fs <*> Concurrently as =
    Concurrently (fmap (\(f, a) -> f a) (concurrently fs as))

instance Monad Concurrently where
  return = pure

  Concurrently a >>= f =
    Concurrently (a >>= runConcurrently . f)

concurrently :: IO a -> IO b -> IO (a, b)
concurrently left right = concurrently' left right (collect []) where
  collect [Left a, Right b] _ = pure (a, b)
  collect [Right b, Left a] _ = pure (a, b)
  collect xs m = do
    e <- takeMVar m
    case e of
      Left ex -> throw ex
      Right r -> collect (r:xs) m

concurrently'
  :: IO a
  -> IO b
  -> (MVar (Either SomeException (Either a b)) -> IO r)
  -> IO r
concurrently' left right collect = do
  done <- newEmptyMVar
  mask $ \restore -> do
    let run a r = restore (a >>= putMVar done . Right . r)
                    `catch` (putMVar done . Left)
    lid <- fork (run left  Left)
    rid <- fork (run right Right)
    let stop = killThread rid >> killThread lid
    r <- restore (collect done) `onException` stop
    stop
    pure r
```

Listing 71: The implementation of the `Concurrently` type.

5.10. Evaluation

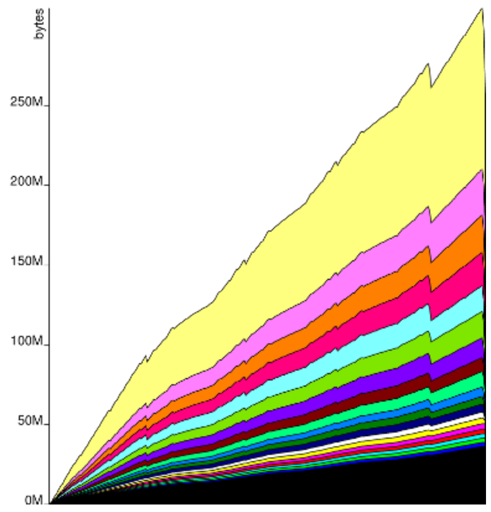
A tool is effectively useless if it is too difficult to use. The main obstacle to the use of Déjà Fu is existing libraries which use `IO`; a programmer cannot simply use `liftIO` everywhere, without sacrificing completeness in all but simple cases. Ideally, existing libraries would be modified to support the `MonadConc` abstraction. However, this does not seem a likely short-term goal, and so a more promising way to approach the problem is to provide alternatives to existing libraries. As adapting code to `MonadConc` is often straightforward, as seen in the monad-par case study (§5.9.1), this is a viable solution.

Users Although Déjà Fu is a small one-man project, it does have some users and contributors. Ten users have opened issues on the GitHub issue tracker; a further three have asked me for help over IRC and email; and ten have made small contributions. Two features came directly from user requests, motivated by performance concerns in large test cases: (1) random scheduling, which we discuss further in Chapter 6; and (2) the ability to discard uninteresting results or execution traces as they are discovered, before evaluating the predicate at the end. This second feature can have a significant impact. If you are interested in a particular failure, it is much better to discard those results which *do not* exhibit the failure as they are discovered, rather than keep them around in memory until the end. Figure 14 shows the effect of this on a simple test of `MVar` contention.

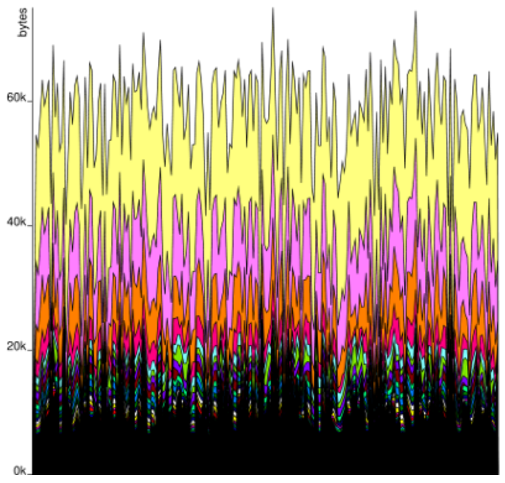
Tweag I/O[5], a research and development company based in Paris, are using Déjà Fu as part of their work on a distributed system for the SAGE project[6], which is investigating storage systems for future supercomputers. They cannot share details of their work for commercial reasons, but one developer had this to say (emphasis mine):

> Regarding the test case: we have an implementation of a distributed storage cluster, with possibly many nodes and parallel requests. The storage system itself is composed of several layers, which can be stacked on top of one another. There is a lot of asynchronicity involved. As per the tests themselves, I am testing parallel requests over different objects, etc.
> *I'd like to add that dejafu tests are by far the most reliable tests in our suite, in my experience - I am yet to see a concurrency bug that they didn't spot, while some other tests missed them!* (Tweag I/O developer 2017)

[5] http://www.tweag.io/
[6] http://www.sagestorage.eu/



(a) Keeping all results and traces in memory.



(b) Discarding all traces, just keeping results.

Figure 14: Heap profiles of a test case for `MVar` contention. Note the dramatic difference: 250M without discarding vs 60k with. These plots are intended to be viewed with colour.

5.10.1. Richness of the Abstraction

As we noted in Section 5.1.2, there are some areas which we do not currently aim to support with Déjà Fu:

- Blocking a thread until a file descriptor becomes available, as this introduces an additional source of nondeterminism.
- Throwing an exception to a thread if it becomes deadlocked, as we cannot reliably detect deadlock involving only a subset of threads without support from the garbage collector.
- Querying which capability (OS thread) a Haskell thread is running on, as this introduces an additional source of nondeterminism.

These three areas are out of scope because we believe that the desire for this functionality does not outweigh the implementation cost. We will look for a way, if that belief changes.

Introducing additional sources of nondeterminism into an SCT model is difficult. Simply introducing additional threads to model the nondeterminism can cause an explosion of schedules tested, which is unsatisfactory. `MonadConc` will always be limited to what Déjà Fu can reasonably support.

```
instance MonadLogger (LoggerT STM) where -- ...
instance MonadLogger (LoggerT IO) where -- ...
```

Listing 72: Concrete instances for a typeclass-based logging abstraction.

```
instance MonadSTM  m => MonadLogger (LoggerT m) where -- ...
instance MonadConc m => MonadLogger (LoggerT m) where -- ...
```

Listing 73: Overlapping instances for a typeclass-based logging abstraction.

However we can still push back the boundaries of what Déjà Fu supports. Bound threads, Haskell threads which always run on the same unique OS thread, were once out of scope as well. This made it impossible to use some C libraries with `MonadConc`. Now we have a prototype implementation[7] which shows promise, and which we intend to release.

5.10.2. Writing and Porting Class-polymorphic Code

We saw in the Par monad example that porting complex code to the `MonadConc` abstraction is not necessarily a difficult process. However, this is not always the case. Recently a user tried to port a logging abstraction they made to `MonadConc`. Expressing this with `MonadConc` and `MonadSTM` is not straightforward, as constraints do not factor into instance selection. So the instances in Listing 73 overlap, and are illegal in standard Haskell.

After some work, we introduced new `IsSTM` and `IsConc` types to disambiguate between the two cases, and ended up with Listing 74. The amount of effort required to arrive at this solution led to the user questioning whether classes like their `MonadLogger` were even a useful abstraction at all[8]! This is a good topic to think about, but it should not be prompted by the effort of trying to use Déjà Fu.

So while porting code to the `MonadConc` typeclass is often simple when dealing with datatype and function definitions, it can be more complicated when dealing with typeclasses. It is not clear what can be done to improve this matter.

[7] https://github.com/barrucadu/dejafu/issues/126

[8] https://www.reddit.com/r/haskell/comments/7b1fbk/do_mtlstyle_effect_classes_really_pull_their/

```
instance MonadSTM  m => MonadLogger (LoggerT (IsSTM  m)) where -- ...
instance MonadConc m => MonadLogger (LoggerT (IsConc m)) where -- ...
```

Listing 74: Polymorphic instances for a typeclass-based logging abstraction.

5.10.3. Library Alternatives

There are popular Haskell libraries specifically for concurrency. One of these is the async library (Marlow 2017), part of which we looked at as a case study (§5.9.3), for expressing asynchronous computations. This library is intended to be a higher-level and safer way of expressing asynchronous computations with guaranteed cleanup than using threading, mutable state, and asynchronous exceptions directly.

Our concurrency library, which provides `MonadConc`, includes an alternative to async using `MonadConc`. There is a test suite using Déjà Fu. The test suite for async itself just runs most tests a single time, although one of them is run 1000 times. Using Déjà Fu here to automatically seek out interesting schedules is a much more principled approach then repetition and hope. Not all features of async are supported, however. As we do not currently support bound threads, functions that use them have been omitted.

This is just one library, and providing an alternative library that people will have to switch to is far from optimal. However, until library authors start to use Déjà Fu and `MonadConc` directly, such alternatives will be needed to answer the question 'Why should I use this if I cannot use it with anything I use already?'

5.10.4. Tool Integration

There are two popular libraries for unit testing in Haskell, HUnit (Herington and Hengel 2017) and Tasty (Cheplyaka 2018). From the perspective of the user, the libraries are similar, but from the perspective of the implementer, they have different approaches to integration. The hunit-dejafu (Walker 2018a) and tasty-dejafu (Walker 2018b) packages provide integration with both.

These packages provide analogues of the Déjà Fu functions, but using the interface of the testing frameworks, rather than computing and printing results directly. The test-framework (Bolingbroke 2017) library is also in common use, however it supports integration with HUnit, and so needs no special support of its own.

The Tasty library is more featureful than HUnit, supporting progress reporting, giving a message on success as well as failure, and command-line arguments. The tasty-dejafu package is similar to the hunit-dejafu package and does not make use of these additional features.

5.11. Summary

In this chapter we presented Déjà Fu, our tool for testing concurrent Haskell programs:

- We provide a typeclass abstraction over concurrency. Such an abstraction allows `IO` to be swapped out and replaced with another implementation (§5.2).
- We implemented a model of Haskell concurrency (§5.4), with an empirically derived operational semantics (§5.5), so that we can simulate the behaviour of GHC. Our model includes most of the Control.Concurrent library module, although some operations are out of scope, or have their behaviour changed (§5.1.2).
- We use bounded partial-order reduction (Coons, Madan Musuvathi, and McKinley 2013) with relaxed memory (Zhang, Kusano, and Wang 2015) as the core testing algorithm for Déjà Fu, but also support a controlled random scheduling approach (§5.6).
- We have not attempted a formal proof of correctness of Déjà Fu, but have made an informal correctness argument, noting the limits of how correct Déjà Fu can be. We do have a comprehensive test suite, and check what correctness conditions we can (§5.8).
- We have discussed three case studies, all of which involved applying Déjà Fu to pre-existing code. Such code must be modified to use the Déjà Fu typeclass abstraction, but we have found this to be a straightforward and type-directed process in most cases (§5.9).

Although a commonly reported experience amongst Haskell programmers is that "if it compiles, it works," there are times when it does *not* work. Concurrency is a particularly difficult area to get right, as everyone who has had to move outside the realm of guaranteed determinism will know. By developing Déjà Fu, we hope that concurrency in Haskell will become a little easier to get right.

Context Déjà Fu does not stand alone, it is related to our other contributions:

- Chapter 6 discusses a new random scheduling algorithm for incomplete concurrency testing. The chapter does not directly build on Déjà Fu, but Déjà Fu implements the algorithm.
- Chapter 7 discusses a new property-discovery tool for functions operating on mutable state. The tool directly builds on Déjà Fu in two ways: (1) to discover these properties, and (2) by providing an interface for Déjà Fu to check them.

Chapter 6

Scheduling Algorithms

We have seen that the complete-within-a-bound approach of DPOR is not suitable for some programs. Programs with large state-spaces, or with sources of nondeterminism other than the scheduler, pose a problem. One way to address this is to sacrifice completeness, and instead explore the space of schedules *randomly*. We may not find all bugs. However we still want to find *most of them*. Benchmarks show that some scheduling algorithms tend to be better at this than others; not all algorithms are created equal. In this chapter we discuss two different randomised scheduling algorithms (§6.1) and then propose a new one based on a *weighted* random selection of threads (§6.2). We show that our proposed algorithm performs favourably in a comparison of standard benchmark programs (§6.3), and evaluate the approach (§6.4).

6.1. Concurrency Testing with Randomised Scheduling

Concurrency testing using randomised scheduling works by repeatedly executing a concurrent program, exploring a particular schedule on each execution. Unlike systematic testing, these algorithms are incomplete in general, and little effort is made to avoid repetition of schedules. In this section we present two approaches to randomised scheduling.

Controlled random scheduling A controlled random scheduler uses a pseudorandom number generator to choose threads to execute. At each scheduling point, a runnable thread is selected at random. This thread is then executed until the next scheduling point. Like any controlled scheduling technique, the executed schedule can be recorded and replayed. Additionally, a controlled random scheduler can be used on programs that exhibit nondeterminism beyond scheduler nondeterminism, although in this case schedule replay will be unreliable (Thomson, Donaldson, and Betts 2016).

Probabilistic concurrency testing The PCT algorithm (Burckhardt et al. 2010) is a randomised algorithm with a probabilistic guarantee of finding bugs. Key to its design is a notion of the *depth* of a bug: the minimum number of scheduling decisions required to exhibit it. The intuition behind PCT is that most concurrency bugs can be exhibited with just a few scheduling decisions being made in the correct places, or the incorrect places depending on your point of view. If PCT makes those decisions correctly, it finds the bug regardless of what any other scheduling decisions were.

On each execution, PCT focuses on finding bugs of a particular depth d. PCT uses a priority-based scheduler where the highest-priority runnable thread is scheduled at each point. d priority change points are distributed randomly through the execution, which is why we must know the length, k, of the program. When execution reaches a change point, the priority of the currently executing thread is changed to a low value.

The algorithm proceeds as follows; given a program with at most n threads and at most k steps, choose a depth d, then:

1. Randomly assign each of the n threads a distinct priority value from $\{d, d{+}1, \ldots, d{+}n\}$. The lower priority values $\{1, \ldots, d{-}1\}$ are reserved for change points.
2. Randomly pick integers $c_1, \ldots, c_{d\text{-}1}$ from $\{1, \ldots, k\}$. These will be the priority change points.
3. Schedule threads strictly according to their priorities: never schedule a thread if a higher priority thread is runnable. After executing the c_i-th step $(1 \leq i < d)$, change the priority of the thread that executed the step to i.

In a single run of a program with n threads and k steps, PCT finds a bug of depth d with probability at least $1/nk^{d\text{-}1}$.

Dynamic partial-order reduction The algorithms we discuss in this chapter are *alternatives* to DPOR. They are useful when the program is too big for a complete approach, even with schedule bounding, and so exploring a representative sample of executions is all we have left.

There is another approach: combining randomisation with DPOR. Sen 2007 proposes the RAPOS algorithm, which uses DPOR to prune the search space, and then explores a random subset of the partial orders discovered. They show that this approach samples different partial orders more uniformly than a simple randomised scheduler, and so it is likely to discover faulty executions more reliably.

6.2. Weighted Random Scheduling and Swarm Testing

We now present *swarm scheduling*, our new algorithm for finding concurrency bugs with controlled scheduling. The algorithm is inspired by *swarm testing* (Groce et al. 2012), an approach to finding bugs using random testing more effectively. Swarm testing makes the observation that, in a random testing tool with many available choices, a uniform distribution is unlikely to discover bugs which correspond to an unfair result:

> As a simple example, consider testing an implementation of a stack ADT that provides two operations, push and pop. [...] To make this example more interesting, imagine the stack implementation has a capacity bug, and will crash whenever the stack is required to hold more than 32 items. (Groce et al. 2012)

The authors then argue that tests generated by randomly interleaving push and pop operations are unlikely to produce a stack with more than 32 items, as items would tend to be popped as quickly as they are pushed. The proposed alternative is, rather than having a *single* distribution for all tests, generate *multiple* distributions to encourage greater variety.

We transfer this idea to the context of scheduling algorithms by observing that controlled random scheduling is much like using a single distribution to generate random tests. So instead, we assign a randomly chosen *weight* to each new thread as it is created, and schedule threads with a weighted random selection. This approach is similar to PCT, but less deterministic: PCT will always schedule the highest-weighted runnable thread, whereas our approach is most likely to, but may not. We can also introduce change points, as in PCT, where we assign the currently executing thread a new weight.

With weight change points included, the algorithm is as follows: given a program with at most k steps, choose a range of weights $[w_{min}, w_{max}]$ and a bound d, then:

1. Assign the initial thread a weight from $[w_{min}, w_{max}]$.
2. Randomly pick integers $c_1, \dots, c_{d-1}$ from $\{1, \dots, k\}$. These will be the weight change points.
3. Schedule threads by a weighted random selection: at each scheduling point use the weights of the enabled threads to construct a nonuniform distribution and pick a thread to run until the next scheduling point. As new threads are created, assign a weight from $[w_{min}, w_{max}]$. After executing the c_i-th step $(1 \leq i < d)$, change the weight of the thread that executed the step to a random value from $[w_{min}, w_{max}]$.

Déjà Fu has a Haskell implementation of swarm scheduling and of controlled random scheduling using a uniform distribution, where the latter is treated as a special case of swarm scheduling where $w_{min} = w_{max}$.

6.3. Comparing Bug-finding Ability

We shall now see how swarm scheduling compares with PCT in terms of bug-finding ability. We use a published collection of benchmark programs, SCTBench (Thomson, Donaldson, and Betts 2016; Thomson, Donaldson, and Betts 2014), and a modified version of the Maple tool (Yu et al. 2012). Maple is a tool for testing Linux programs which use POSIX threads (IEEE 1995). We use Maple because the prior work using SCTBench also does. We want any difference in algorithm performance to be due to the algorithms themselves, not because of any difference in how the host tool works.

Maple comes with a PCT implementation using Linux real-time thread priorities, but we use the modified version from Thomson, Donaldson, and Betts 2016 instead. This modified version differs from the standard PCT algorithm slightly. PCT does not directly handle yielding threads: if the highest-priority runnable thread is in a busy-wait loop, it may yield until some condition holds. Immediately scheduling the thread again after it yields would lead to a nonterminating execution. The original PCT implementation uses heuristics to determine if a thread is not making progress, and to lower its priority (Burckhardt et al. 2010). In the implementation we use, the priority of a yielding thread is changed to the lowest priority.

6.3.1. Benchmark Collection

SCTBench (Thomson, Donaldson, and Betts 2016; Thomson, Donaldson, and Betts 2014) is a collection of pthread programs amenable to concurrency testing by controlled scheduling. All the programs are deterministic, other than scheduler nondeterminism. In total there are 49 benchmark programs. SCTBench is assembled from several other sets of benchmarks, so there is some variety in the programs:

- Buggy versions of aget (a file downloader) and pbzip2 (a compression program).
- A set of test cases for a work-stealing queue.
- Examples used to test the ESBMC tool (Cordeiro and Fischer 2011), an SMT-based model checker for concurrency.
- Examples used to test the INSPECT tool (Y. Yang, Chen, and Gopalakrishnan 2008), a concurrency testing tool for instrumented C programs.

- A buggy lock-free stack implementation.
- A test case exposing a bug in the ctrace (McPherson 2004) concurrency debugging library.
- Buggy versions of a content similarity search tool and online clustering tool.
- Three benchmarks exposing bugs in Mozilla SpiderMonkey (Eich and Mozilla Foundation 1996) and the Mozilla Netscape Portable Runtime Thread Package (Mozilla Foundation 1996).
- The SPLASH-2 programs (Woo et al. 1995).

6.3.2. Experimental Method

We aim to compare swarm scheduling, using a variety of parameters, with PCT and controlled random scheduling. We do not consider the other algorithms used in the prior SCTBench work, or PCT with a bound other than $d = 3$, as setting $d = 3$ was found to give the best results in terms of bug-finding ability (Thomson, Donaldson, and Betts 2016).

In total, we try 5 algorithm-parameter variants:

- Controlled random scheduling.
- PCT with $d = 3$.
- Swarm scheduling with $d \in \{1, 2, 3\}$.

For each variant, we run each benchmark with a limit of 10,000 executions. We use a schedule limit rather than a time limit, as many factors can influence timing results, and they are not readily comparable or reproducible. Number of executions required, however, is an intrinsic property of the benchmark program, the testing algorithm, and any parameters.

We were fortunate enough to have access to the scripts used by Thomson, Donaldson, and Betts 2016 to run the benchmarks, which greatly simplified experimentation. Each benchmark goes through each of the following phases of testing:

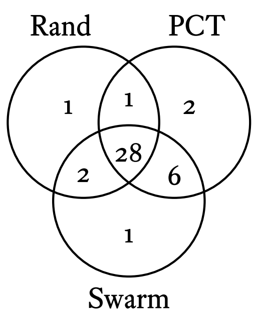
Data race detection phase It is sound to only consider scheduling points before synchronisation operations as long as execution aborts with an error as soon as a data race is detected (Madanlal Musuvathi and Qadeer 2008). This greatly reduces the number of schedules that need to be considered. However, the benchmark programs contain many benign data races (Thomson, Donaldson, and Betts 2016), so this condition is too strict. As in prior work (Thomson, Donaldson, and Betts 2016; Thomson, Donaldson, and Betts 2014; Yu et al. 2012) we address this problem by performing dynamic data race detection first, to identify a subset of load and store operations which are known to be racey, which are then treated as visible synchronisation operations during testing. This process is nondeterministic, so we run it ten times for each benchmark.

Controlled random scheduling phase We run each benchmark 10,000 times using Maple's controlled random scheduler. Although this approach was found to be inferior to PCT (Thomson, Donaldson, and Betts 2016), we include it so we have a naïve baseline for evaluation purposes.

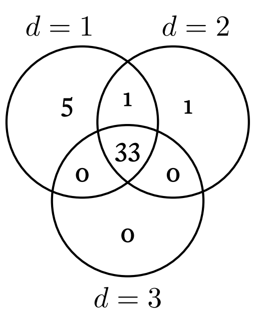
PCT testing phase PCT requires parameters n, the maximum number of threads; k, the maximum number of steps in the execution; and d, the bug depth. We fix $d = 3$, and use estimates for n and k found by Thomson, Donaldson, and Betts 2016. These estimates were obtained by making an initial estimate and then executing PCT with $d = 3$, on the assumption that this would increase interleaving, and counting steps from when the first thread launches the second. We run each benchmark 10,000 times using its estimated n and k values.

Swarm scheduling phase Swarm scheduling requires parameters w_{min}, the minimum weight; w_{max}, the maximum weight; k, the maximum number of steps in the execution; and d, the number of weight change points to include. We want to encourage executions with very unequal thread weights, and so pick $w_{min} = 1$ and $w_{max} = 50$, giving significantly more weights than most benchmarks have threads. We use the same k values as in PCT. We then perform multiple runs of swarm scheduling: we perform 10,000 executions of each benchmark program for each d, using its estimated k value.

Note on randomness For a given benchmark, we can use the average number of schedules needed to expose a bug (10,000 $\div$ the number of buggy schedules) to compare techniques. The exact value is dependent on the initial seed, but we would expect it to become stable as the number of executions is increased (Thomson, Donaldson, and Betts 2016).



(a) Basic algorithms



(b) Weight changes

Figure 15: Overlap of bugs found by each scheduling algorithm.

6.3.3. Experimental Results

We conducted our experiments using an Ubuntu 12.04 virtual machine, and a modified version of Maple based on the last commit from 2012[1]. Listing 96 in Appendix B shows the core of our C++ swarm scheduling implementation. Other than the addition of swarm scheduling, the code is unchanged.

The Venn diagrams in Figure 15 show the relative bug-finding ability of each algorithm. Figure 15a summarises the bugs found by controlled random scheduling, PCT $d = 3$, and swarm scheduling $d = 0$ (which we now call "Swarm" for brevity). Swarm performs comparably with PCT. Figure 15b shows the effect of introducing weight change points, which seems to harm performance.

As the techniques we have considered are nondeterministic, it is interesting to consider their average-case behaviour. Figure 16 shows the aggregate behaviour of the algorithms across all benchmarks as the number of executions increases. Swarm outperforms PCT at first, but PCT catches up as the number of executions increases.

The plots in Figure 17 show the average number of executions to find a bug across all benchmarks; as expected, the average number of executions to find *any* bug rapidly converges, however the average number of schedules to find a *unique* bug does not. This is due to two factors: (1) the number of bugs is finite; and (2) some bugs may be out of reach of a particular algorithm. Table 8 shows the final values. We can see that PCT and Swarm are almost identical, and both significantly improve upon random scheduling when it comes to finding unique bugs.

[1] The same environment as Thomson, Donaldson, and Betts 2016, available at `https://github.com/mc-imperial/sctbench`

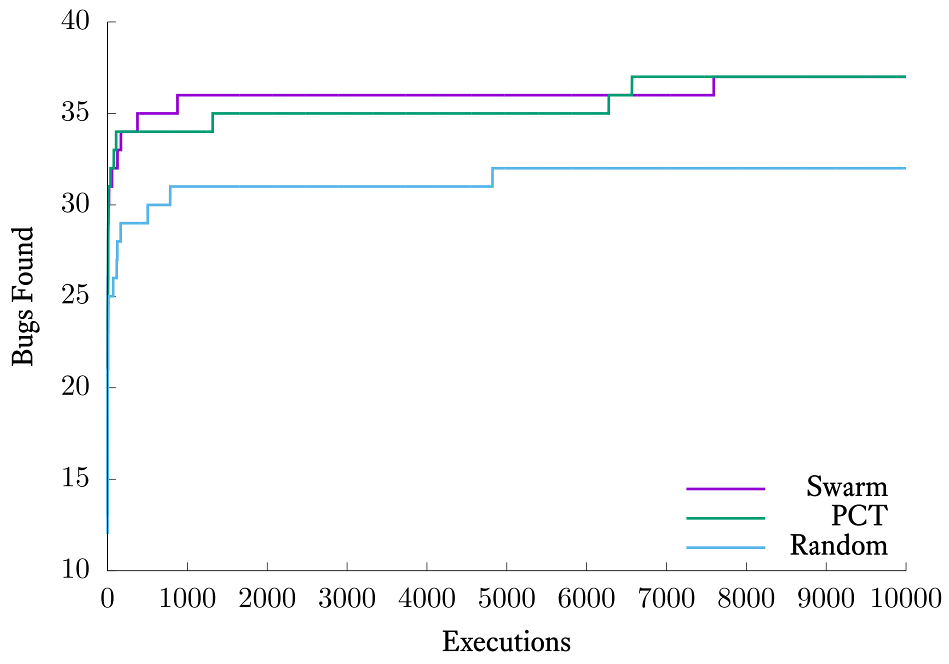


Figure 16: The number of bugs found by each algorithm across all benchmarks. This plot is intended to be viewed with colour.

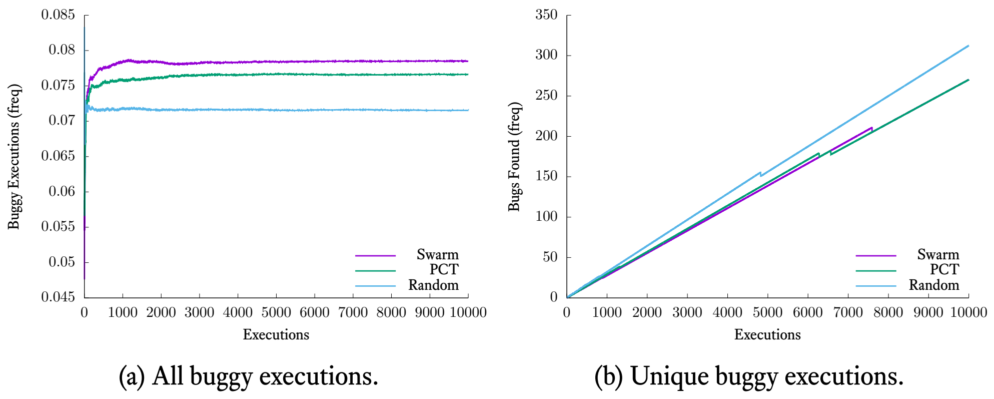


Figure 17: The average number of executions to expose a bug across all benchmarks. These plots are intended to be viewed with colour.

Algorithm	Any Bug	A Unique Bug
Random	0.071	312.5
PCT	0.077	270.3
Swarm	0.078	270.3

Table 8: Average number of executions needed to find a bug.

6.4. Evaluation

Testing concurrent programs by using a swarm of randomly generated weighted distributions appears to work well in practice. The simplest variant, which we simply call Swarm, uses no weight change points. So for Swarm, the k and d parameters are irrelevant, which means the user does not need to know the maximum length of their program execution in advance. Although it is disappointing that Swarm does not improve upon PCT, in terms of bug-finding it does perform comparably.

A significant advantage of the Swarm method over PCT is that it does not require the programmer to first determine the maximum length of an execution, which can be difficult to estimate.

Randomised stride Another algorithm with a similar motivation is the randomised stride algorithm (Abdelrasoul 2017), which outperforms PCT (and so our Swarm algorithm). This is a recent algorithm which was not yet published when we began this work. Like PCT, randomised stride derives its power from knowledge of the underlying program: in this case, an estimate of the length of each thread. Although Swarm performs worse than randomised stride, we believe it still has a place in the concurrency testing arsenal as an effective algorithm which does not require first estimating or measuring some parameter of the program under test.

Why are weight changes bad? Figure 15b shows that, although using one weight change point results in finding an additional bug, bug-finding ability rapidly degrades as more points are introduced. We suspect that this is because frequent weight changes compromise the intuition for why weighted random testing is effective at all. We believe that weighted random testing is effective because unfair scheduling causes threads to make progress at different rates, leading to interleavings which the programmer is unlikely to have considered. However, by changing weights, a thread which was previously making rapid progress may suddenly slow down, allowing other threads to catch up. In the extreme, if we introduce a weight change after every scheduling point, we have simply produced a complicated version of uniform random scheduling.

6.5. Summary

In this chapter we presented swarm scheduling, our scheduling algorithm for finding faults in concurrent programs:

- We propose that unfair schedules are likely to reveal concurrency bugs more effectively than fair schedules. Unfair schedules cause different threads to make progress at different rates, resulting in interleavings the programmer is unlikely to have considered (§6.4).
- Swarm scheduling uses weighted random scheduling, with optional weight change points. However, frequently changing the weights causes weighted scheduling to degrade to an effectively uniform scheduling (§6.2).
- We find that one parameterisation of the swarm scheduling algorithm performs as well as PCT (Burckhardt et al. 2010), despite not knowing anything about the program under test. We argue that this makes swarm scheduling easier to use, while giving just as good results (§6.3).

The complete systematic approach to testing is not necessarily suitable for all programs. Programs with large state-spaces, or programs which have sources of nondeterminism other than the scheduler, cannot readily be tested with such techniques. For such programs, random testing is commonly used. By introducing swarm scheduling, we hope that effective random testing of concurrent programs will become simpler.

Context Swarm does not stand alone, it is related to our other contributions:

- Chapter 5 uses swarm scheduling to provide its random testing mode. Complete testing is the default. Swarm scheduling was added for cases where the state-space is too large to systematically explore.

Chapter 7

CoCo: Discovering Properties Automatically

In this chapter we present and evaluate CoCo, our tool for finding properties of equivalence and refinement between concurrent Haskell expressions with shared state. We first discuss what properties for concurrency even are (§7.1), and then demonstrate the use of the tool with an illustrative example (§7.2). We explain how properties are discovered (§7.3) and argue the correctness of our approach (§7.4). We then present two further examples (§7.5). Next we discuss how CoCo properties can be incorporated into a Déjà Fu test suite (§7.6). Finally, we present conclusions and evaluate the approach (§7.7).

This chapter is derived from our previous work Walker and Runciman 2017 and Walker and Runciman 2018.

7.1. Concurrency Properties

CoCo works by generating program fragments and then comparing their effects. Unlike tools for discovering properties of *sequential* programs, such as QuickSpec (Smallbone et al. 2017) and Speculate (Braquehais and Runciman 2017b), CoCo cannot simply compare the results of single executions. In general, concurrent programs are nondeterministic, so we must consider all possible behaviours when determining if two terms are equivalent in some sense.

Properties We formulate our properties in terms of *observational refinement* (He, Hoare, and Sanders 1986). In general, one term is an observational refinement of another if the *visible effects* of the first term are a subset of those of the second term. The properties that we report are of the form `A === B`, meaning that `A` and `B` are observationally equivalent; and `A ->- B`, meaning that `A` is a strict observational refinement of `B`.

```
evalSigWithSeed :: (MonadConc m, Ord o)
  => (x -> ConcT m s)        -- ^ Create a new instance of the state.
  -> (s -> x -> ConcT m o)   -- ^ The observation to make.
  -> (s -> x -> ConcT m ()) -- ^ Perform some concurrent interference.
  -> (s -> ConcT m ())       -- ^ The expression to evaluate.
  -> x                       -- ^ Pure value used to initialise the state.
  -> m (Set (Maybe Failure, o))
```

Listing 75: The `evalSigWithSeed` function, repeated from Section 5.1.1. Runs a concurrent program and returns a set of observations and possible failures.

CoCo uses Déjà Fu to find the results of generated program fragments, and then tests these relations. Déjà Fu provides support for testing if two programs are observationally equivalent (§5.1.1), CoCo extends this to also support observational refinement.

Visible effects CoCo does not consider all possible effects, we require the programmer to provide an *observation function* for a nominated piece of shared state. It is the observations produced by this function which we compare to determine if one term is a refinement of another.

Concurrent interference If CoCo reports that two program fragments are equivalent, then they should be interchangeable. However, if we did not consider concurrent interference, they may not be. In addition to the observation function, CoCo requires the programmer to supply an effectful *interference function*, which is executed concurrently during testing. By supplying different sorts of interference, the programmer can see how the functions under test behave in different concurrent contexts.

What a property means A CoCo property stating `A === B` can be read as "when executed concurrently with a user-specified interference function `I`, and followed by a user-specified observation function `O`, the result of `O` for both terms is the same." This is a weaker requirement than how observational equivalence is usually used, where `A === B` means that you can replace `A` with `B` anywhere. The weaker claim of CoCo is due to the tool being based on testing, rather than proof.

Like Déjà Fu, CoCo does not reason about the terms it is given, it runs them and examines the results. Listing 75 gives the function at the heart of CoCo, which we saw in Section 5.1.1. Here the first three arguments—the function to initialise the state, the observation function, and the interference function—are supplied by the programmer. The expression to evaluate and the pure "seed" value are generated by CoCo.

```
putMVar  :: MVar Concurrency Int -> Int -> Concurrency ()
takeMVar :: MVar Concurrency Int -> Concurrency Int
readMVar :: MVar Concurrency Int -> Concurrency Int
```

Listing 76: Type signatures for `MVar` operations in CoCo.

Despite its limitations, we believe that CoCo properties can still provide a useful glimpse into the behaviour of a concurrent program. We motivate this throughout the chapter with examples.

7.2. An Illustrative Example

Let us now show an example use of CoCo for Haskell `MVars`. Recall that an `MVar` is a mutable memory cell which may be *full* or *empty*. We now examine three basic operations over `MVars`: put, take, and read. To *put* is to block until the `MVar` is empty and then set its value. To *take* is to block until the `MVar` is full, remove its value, and return the value. To *read* is to *take*, but without emptying the `MVar`. Each function has a non-blocking *try* variant, which returns an indicator of success.

Allowing shared values of type `Int`, we obtain the type signatures in Listing 76. Here `Concurrency` is an implementation of the `MonadConc` typeclass from Chapter 5, provided by CoCo. The `MVar` type is an abstract type defined in `MonadConc`, with the concrete type determined by the monad used. In this case the concrete type is defined as part of Déjà Fu.

Signatures When we use CoCo, we must provide four things: (1) the functions and values which may appear in properties; (2) a way to initialise the state; (3) an observation function; and (4) an interference function.

We call this collection of programmer-supplied definitions the *signature*. Listing 77 shows a signature for `MVar` operations. The initialisation function constructs an empty or a full `MVar`. The interference function simply stores a new value. The observation function takes a snapshot of the state. The `backToSeed` function is used to check whether the state has been changed: if the original and final seed values are the same, the state is unchanged.

It is essential to provide an initialisation function which gives a representative collection of states, and an interference function which can disrupt the functions of interest. If our initialisation function only produced a full `MVar`, we could find properties which do not hold when the `MVar` is empty. Because our interference function only writes to

```
type C = Concurrency

sig :: Sig (MVar C Int) (Maybe Int) (Maybe Int)
sig = Sig
  { initialise  = maybe newEmptyMVar newMVar
  , expressions =
    [ -- example 1
      lit "putMVar"  (putMVar  :: MVar C Int -> Int -> C ())
    , lit "takeMVar" (takeMVar :: MVar C Int -> C Int)
    , lit "readMVar" (readMVar :: MVar C Int -> C Int)
    ]
  , backgroundExpressions =
    [ -- example 2
      lit "tryPutMVar" (tryPutMVar :: MVar C Int -> Int -> C Bool)
    ]
  , interfere  = \v _ -> putMVar v 42
  , observe    = \v _ -> tryReadMVar v
  , backToSeed = \v _ -> tryReadMVar v
  }
```

Listing 77: A CoCo signature for `MVar`s holding `Int`s.

```
(1)    readMVar @  ===  readMVar @ >> readMVar @
(2)    readMVar @  ->-  takeMVar @ >>= \x -> putMVar @ x
(3)    takeMVar @  ===  readMVar @ >> takeMVar @
(4)  putMVar @ x   ===  putMVar @ x >> readMVar @
```

Listing 78: Some properties CoCo discovers about `MVar`s.

the `MVar`, we may find properties which do not hold when there are multiple consumers. Developing a fuller understanding of the functions under test may require examining the different property-sets found under different execution conditions.

Discovering properties Listing 78 shows the properties which CoCo discovers given the signatures in Listing 77. In this output, `@` is the state argument, which is the `MVar`. For convenience of reference, we have added numbers to the CoCo properties. These numbers are not part of the normal output of the tool.

Property (1) shows that `readMVar` is idempotent; (2) shows that it is not merely a take followed by a put, it is rather a distinct operation; (3) and (4) show that it does not modify the `MVar`, and that it does not block when the `MVar` is full. The terms in Properties (2) and (4) do not have the same types, illustrating a weakness in how CoCo currently works: CoCo does not consider the results of the terms, only their effects.

We see the effect of the interference in (2): with no other producers, this would be an equivalence; it is only when interference by another thread is introduced that the equiva-

Term	Seed	Final State	Deadlocks
Read	`Nothing`	`Just 42`	No
	`Just 0`	`Just 0`	No
Take / Put	`Nothing`	`Just 42`	No
	`Just 0`	`Just 0`	No
		`Just 42`	Yes

Table 9: The behaviours of the terms in Property (2).

```
      readMVar @  ===  readMVar @ >> tryPutMVar @ x
(5)   readMVar @  ===  readMVar @ >>= \x -> tryPutMVar @ x
      readMVar @  ->-  takeMVar @ >>= \x -> tryPutMVar @ x
    putMVar @ x   ===  putMVar @ x >> tryPutMVar @ x1
```

Listing 79: Some more properties CoCo discovers about `MVars`.

lence breaks down and the distinction is revealed. Table 9 shows the possible behaviours in Property (2). This property is a strict refinement because, while the behaviours for the seed value `Nothing` are the same, the behaviours of the left term for the seed value `Just 0` are a strict subset of the behaviours of the right.

Background expressions Sometimes when expressing properties it is necessary to call upon other expressions which are of secondary interest. Such expressions are commonly called *background* expressions. A property is only reported if each term includes at least one non-background expression.

If we include `tryPutMVar`, a non-blocking version of `putMVar`, as a background expression, CoCo discovers the additional properties in Listing 79. Property (5) shows how important the choice of interference function is. The left and right terms are not equivalent: if the interference were to empty a full `MVar` then the right term could restore its original value. As our choice of interference function only produces, rather than consumes, it will never alter the value in a full `MVar`.

The example in this section takes about 1.5 seconds to run, and the output displayed here is the output of the tool, aside from the property numbers. We discuss further performance results in Section 7.7.

7.3. How CoCo Works

A simplified version of our approach is to generate all terms up to some syntactic size limit, compute and store their behaviours, and then find properties by comparing the sets of behaviours of each pair of terms. This would be slow, however. Following the lead of QuickSpec (Claessen, Smallbone, and Hughes 2010; Smallbone et al. 2017) we make three key improvements:

1. We generate *schemas* with *holes*, rather than *terms* with *variables*.
2. We only compute the set of behaviours of the most general term of every schema.
3. We interleave property discovery with schema generation, and aggressively prune redundant schemas.

Why not use QuickSpec's term generator? The main difference between our approach and QuickSpec is how we handle monadic operations. We generate lambda-terms in a restricted setting, but QuickSpec does not do so at all. This means that QuickSpec is unable to generate larger monadic expressions out of smaller ones, unless the user supplies in the signature functions representing all the ways in which they may be combined. CoCo is able to generate the needed lambda functions used in monadic binds.

7.3.1. Representing and Generating Expression Schemas

We can greatly reduce the number of expressions considered by not generating alpha-equivalent ones. Instead of generating an expression like `push @ x >> push @ y`, we will instead generate the expression `push @ ? >> push @ ?` where each `?` is a *hole* for a variable. These expressions-with-holes are called *schemas*. One schema can be instantiated into many *terms* by assigning variable names to groups of holes. The `push`-`push` schema has two semantically distinct term instances: the single-variable and the two-variable cases.

Our expression representation is shown in Listing 80. The `Expr` type is parameterised by a state type `s` and a *hole* type `h`. The state parameter ensures expressions that assume different execution contexts cannot be inadvertently combined. The hole parameter allows for a statically enforced distinction between schemas and terms. Each `Expr` constructor carries around a representation of its type (except the state, which is implicit). In most of the implementation we hide the details of this representation and instead provide *smart constructor* functions to ensure only well-typed expressions can be constructed.

```
data Expr s h
  = Lit  String Dynamic
  | Var  TypeRep (Var h)
  | Bind TypeRep (Expr s h) (Expr s h)
  | Ap   TypeRep (Expr s h) (Expr s h)
  | State

data Var h = Hole h | Named String | Bound Int

type Schema s = Expr s ()
type Term   s = Expr s Void
```

Listing 80: Representation of Haskell expressions.

Schema generation Generating new schemas is straightforward. We give expressions a notion of *size*, corresponding roughly to the size of the `Expr` tree. Schemas are generated in size order. The needed expressions of size 1 are supplied in the user's signature. For larger sizes we combine appropriately sized subexpressions represented by already generated schemas and keep the type-correct ones.

We interleave generation with evaluation and property discovery. In this way we can partition schemas into equivalence classes and use only the smallest of known-equivalent schemas when generating new ones. We do this for both pure and monadic schemas.

Monadic expressions The expressions of most interest to us are *monadic* expressions. Such expressions allow us to combine smaller effects to create larger ones. We simplify this task by taking inspiration from Haskell's do-notation, a syntactic sugar for expressing compound monadic operations in an imperative style, which has explicit variable bindings and makes the sequencing of effects clear. Rather than generating lambda-terms, we use a kind of first-class do-notation where the monadic bind operation binds the result of evaluating the *binder* to holes in the *body*. Restricting ourselves to this case allows us to avoid many of the complexities of generating lambda-terms directly.

For example, the generation of the schema `pop @ >>= \x -> push @ x` proceeds as follows:

1. Combine `pop` and `@` to produce `pop @`
2. Combine `push` and `@` to produce `push @`
3. Combine `push @` and `?` to produce `push @ ?`
4. Combine `pop @` and `push @ ?` to produce both `pop @ >> push @ ?` and `pop @ >>= \x -> push @ x`.

```
readMVar @  ===  takeMVar @ >>= \x -> putMVar @ x
```

Listing 81: A property that holds with no interference.

```
f (? :: Int) (? :: Bool) (? :: Bool) (? :: Int)

f (w :: Int) (x :: Bool) (y :: Bool) (z :: Int)
f (w :: Int) (x :: Bool) (y :: Bool) (w :: Int)
f (w :: Int) (x :: Bool) (x :: Bool) (z :: Int)
f (w :: Int) (x :: Bool) (x :: Bool) (w :: Int)
```

Listing 82: A schema and its term instances.

7.3.2. Evaluating Most General Terms

Time spent evaluating terms dominates the execution cost of CoCo. In the worst case the number of executions needed for a term is exponential in the number of threads, context switches, and blocking operations (Madanlal Musuvathi and Qadeer 2007).

What is more, our term evaluation always involves at least two threads: the term thread executing the term itself, and an *interference thread*. The term thread may fork additional threads. The interference thread is essential to distinguish refinement from equality in some cases. For example, the equivalence in Listing 81 holds only when there is no concurrent producer for the same `MVar`.

To avoid repeated work, we compute the behaviours of all the terms for a schema when it is generated. We annotate each schema with some metadata, including its behaviour-sets, and compare these cached behaviours later when discovering properties. Storing this data is a space cost, but reduces the execution time of some of our test applications from hours to minutes. We present performance measurements in Section 7.7.

Deriving terms from schemas One schema may have many term instances. Listing 82 shows an example of a schema with two holes of one type and two of another. From this one schema, we can produce four semantically distinct terms. We can order the terms by number of distinct variables. The term with the most variables is the *most general term*.

We use a simple reduce-and-conquer algorithm to eliminate holes one type at a time:

1. Pick a type and find the set of all holes of that type.
2. For each partition of the hole-set make a distinct copy of the schema and in each case assign to each subset in the partition a distinct variable name.
3. If there are remaining hole types, continue recursively from (1).

4. Finally, sort the terms by number of distinct variables.

Evaluating terms To compute the behaviours of every term for a schema, we need only consider the most general term. The behaviours of all less-general terms can be derived from the most general by restricting to cases where the variables are equal. For example, given the behaviours of the term `f x y`, we throw away those where `x` $\neq$ `y` to obtain the behaviours of the term `f x x`.

Déjà Fu allows us to make an observation of the final state even if evaluation of the term deadlocks. This is essential, as an operation which deadlocks may have altered the state before blocking.

7.3.3. Property Discovery and Schema Pruning

Not only do we interleave generation with evaluation, we also interleave it with property-discovery. After all schemas of a given size are generated and their most general terms evaluated, we compare each such new schema against all smaller ones to discover equivalences and refinements.

As one schema may correspond to many terms, we may discover many properties between a pair of schemas. In practice, most of these properties are consequences of more general ones, so we solve this problem by pruning the generated properties. Property P_2 is a consequence by property P_1 if (1) both P_1 and P_2 are equivalences or both are strict refinements; and (2) P_1 has a more general allocation of variables to holes.

Smallest schemas To avoid discovering the same property multiple times, we maintain a set of *smallest schemas*. At first, all schemas are assumed to be smallest. If a syntactically smaller schema is a refinement of a larger one, the larger is annotated as 'not smallest.' When generating new monadic binds:

- A schema `S >> T` is only generated if both `S` and `T` are smallest schemas.
- A schema `S >>= \x -> T[x]` is only generated if `T` is a smallest schema.

We also only consider properties `S === T` or `S ->- T` where both `S` and `T` are smallest schemas.

Neutral schemas A schema `N` is neutral if and only if, for all other schemas `S`, these identities hold: `N >> S === S === S >> N`. For example, `readMVar` is not a neutral `MVar` operation, as it may block, but the non-blocking alternative `tryReadMVar` is neutral. A sufficient condition for a schema to be neutral is if its most general term instance is (1) always atomic; (2) never fails; and (3) never modifies the state.

We use a heuristic method based on execution traces to determine if a schema is atomic, and use the seed values to determine if it modifies the state. If a schema is judged to be neutral, we do not use it when constructing larger schemas. If a schema is, falsely, judged to not be neutral then it will be incorporated into larger schemas: adding to the execution cost of CoCo and generating new properties.

Projection to a common namespace We compute the behaviours of each term individually, yet we construct properties from pairs of terms. This poses a difficulty when discovering properties between terms which have free variables: we must decide which free variables in one term to identify with free variables in the other term, if any. So we project the two terms into a common namespace. Each variable in each term is either given a unique name, or identified with a variable in the other term. There may be multiple ways to do this, and CoCo tries them all. We never reduce the number of distinct variables in a term. To do so would only reproduce another term generated from the same schema.

As a pair of terms may have many projections, we may discover many properties between them: at most one for each projection. In practice, most of these properties are consequences of more general ones. We only keep the most general.

7.3.4. The CoCo Algorithm

The CoCo algorithm is similar to that of any other property-discovery tool. The key difference is that we compare sets of behaviours, rather than equality of results. We have some additional complication as we evaluate terms independently, but this is not an inherent aspect of CoCo: one could implement a CoCo which does not cache term results like we do, it would just be slow.

In outline, the CoCo algorithm is:

1. For all sizes 1 to the limit:
    (a) Generate all well-typed schemas of this size.
    (b) For each such schema:
        i. Record the results of evaluating its most general term.
        ii. Check if the schema is neutral and, if so, annotate it.
        iii. For each smaller schema, which is considered 'smallest':

A. Discover properties between the two.
B. Display any such properties to the user.

To implement step 1(b)iii we must keep track of all previously generated schemas. Conveniently, this makes it simple to incorporate schemas provided in the user's signature: we include them in the initial state and thereafter treat them like any other schema.

Property discovery Given two schemas `S` and `T` where `S` is larger than `T`, we discover properties between the two like so:

1. For each term `St` of `S` and `Tt` of `T`:
    (a) For each projection `P` between `St` and `Tt`:
        i. Let `Str` be `St` renamed according to `P`.
        ii. Let `Ttr` be `Tt` renamed according to `P`.
        iii. Check if `Str === Ttr`, `Ttr ->- Str`, or `Str ->- Ttr`.
        iv. In the first two cases, annotate `S` as 'not smallest'.
2. Discard any properties made redundant by a more general one.
3. Return the remaining properties.

Our approach differs from QuickSpec (Claessen, Smallbone, and Hughes 2010; Smallbone et al. 2017) in step 1(a)iii, where we compare sets of term behaviours rather than term results.

7.4. Soundness and Completeness

Correctness for CoCo states that only true properties are reported, and that all true properties where the terms involved fit into the size bound are reported. As with Déjà Fu, we do not attempt a proof of formal correctness for the CoCo implementation.

Soundness There are two potential sources of unsoundness in CoCo. Firstly, properties are only checked for a finite number of cases; and secondly Déjà Fu is used to find the possible results of a term, which is incomplete by default.

We can increase confidence in the correctness of CoCo properties by increasing the number of test cases, but the fundamental problem remains. CoCo is not a model checker, and it is always possible that the next input tried after we ceased testing would have been a counterexample. This problem is made worse by the need to achieve acceptable performance. Evaluating terms takes time, so to improve performance we wish to minimise the number of evaluations, and so the number of distinct inputs each term is tried with. If a term takes a wide data type as a parameter, there may be constructors of this data type which are simply not tested at all.

The incompleteness of Déjà Fu can be solved with additional time. Déjà Fu offers a complete mode. CoCo does not use it because it is typically slower than the incomplete testing, but as the terms we generate are small this may not be a problem in practice. CoCo could be changed to use the complete testing, which would ensure that all possible behaviours of a term, for each set of inputs considered, are found.

Completeness There are two potential sources of incompleteness in CoCo. Firstly, some schemas are discarded when generating new ones; and secondly, properties are thrown away which are judged to be consequences of another.

A property between two terms cannot be discovered if either term is simply not generated. When generating new schemas, CoCo tries all appropriately sized pairings of previously generated schemas and keeps the type-correct ones. If our type checker is incorrect, then valid schemas may be thrown away. We do not support class polymorphism at all, so terms which are only well-typed due to typeclass use will be discarded.

We do not require CoCo to report properties which are implied by another. As many terms correspond to the same schema, and many projections correspond to the same pair of terms, in general there will be many such redundant properties. We use two syntactic criteria to determine whether one property implies another, rather than attempting any deeper logical analysis.

A property between terms `A1` and `B1` is made redundant by another property between `A2` and `B2` if both properties are strict refinements or both are equivalences, and:

- If `A1` and `A2` are both renamings of the same term `A` (similarly for `B1` and `B2`), and the renaming which generates `A2` and `B2` is more general than the renaming which generates `A1` and `B1`.
  For example, the property `f x === g x y` is made redundant by `f x === g y z`, as the latter has a more general renaming,
- If `A1` and `A2` are both terms generated from the same schema `A` (similarly for `B1` and `B2`), and `A2` is more general than `A1`, and `B2` is more general than `B1`.
  For example, the term `f x x` is less general than `f x y`, so `f x x ->- g x x` is made redundant by `f x y ->- g x y`.

```
newtype LockStack m a = LockStack (MVar m [a])

push :: MonadConc m => a -> LockStack m a -> m ()
push a (LockStack v) = modifyMVar v (\as -> pure (a:as, ()))

pop :: MonadConc m => LockStack m a -> m (Maybe a)
pop (LockStack v) = modifyMVar v (\as -> (drop 1 as, listToMaybe as))

peek :: MonadConc m => LockStack m a -> m (Maybe a)
peek (LockStack v) = fmap listToMaybe (readMVar v)
```

Listing 83: A lock-based mutable stack.

```
(6)  peek @  ->-  push x @ >> pop @
(7)  peek @  ->-  (push x @) ||| (pop @)
(8)  peek @  ->-  pop @ >>= \m -> whenJust push @ m
```

Listing 84: Some properties CoCo discovers about the `MVar` stack.

So we consider separately (1) the identification of variables between terms, and (2) the mapping of holes to single variables.

7.5. Case Studies

We discuss the process and results of applying CoCo to two concurrent data structures: mutable stacks and semaphores. We chose these as they are common primitives used in the implementation of concurrent algorithms.

7.5.1. Concurrent Stacks

Lock-based stacks Mutable stacks are commonly used for synchronisation amongst multiple threads, for example see Dodds, Haas, and Kirsch 2015. Listing 83 shows a simple mutable stack is just an immutable list inside an `MVar` shared variable.

Listing 84 shows the properties that CoCo discovers with our definitions, where the initialisation function constructs a stack from a list, the observation function converts it back to a list, and the interference function sets the contents of the stack to a given list. Here `whenJust` is defined as `\f s -> maybe (pure ()) (`f` s)` and `|||` is concurrent composition. Property (6) may seem surprising: the left term returns the top of stack whereas the right term returns the value pushed. Remember that CoCo does not consider equality of results when determining properties, only the effect on the state. Property (7) is a consequence of (6). Property (8) is analogous to the `readMVar` properties presented in Section 7.2, as we might expect given how the stack operations

```
push2 x1 x @  ->-  push x @ >> push x @
```

Listing 85: A property about an incorrect function.

```
            peek @  ->-  push x @ >> pop @
            peek @  ===  pop @ >>= \m -> whenJust push @ m
          push x @  ===  pop @ >> push x @
(9)      push x1 @  ===  push2 x x1 @
(10)     push x1 @  ===  push x @ >> push x1 @
whenJust push @ m   ===  whenJust (push2 x) @ m
```

Listing 86: Changing the observation function to `peek` changes the properties discovered.

are defined.

Buggy functions Suppose we add an *incorrect* `push2` function, which is meant to push two values atomically, but which only pushes the second value twice.

CoCo finds the property in Listing 85. As this is a strict refinement, we now know that `push2` is more deterministic in some way than two `push`es. As we know that the composition of two `push`es is not atomic, this strongly suggests that `push2` is. We can also see the effect of `push2` on the state, and that it is incorrect!

Choice of observation As CoCo uses a programmer-supplied observation function in its property-discovery process, the programmer can supply different observations to discover different properties. By changing the observation of our stack from list equality to `peek`, we discover a new collection of properties, shown in Listing 86. Here we have fixed the `push2` function to behave correctly and also removed `|||` from the signature. Properties (9) and (10) show the power of supplying a custom observation function: in the left and right terms, the stack states are *not* equal. In both (9) and (10) the left term increases the stack depth by one, and the right by two. We now see that `push2` leaves its second argument on the top of the stack. We could not directly observe this before, as a single push would leave the stack sizes out of balance. Throwing away unnecessary details, in this case the tail of the stack, allows us to see more than we previously could.

It is important to bear in mind that there is no *best* observation to make, no *best* interference to consider, and no *best* set of properties to discover. Each choice of observation and interference will reveal something about the functions under test. By considering different cases, we can arrive at a fuller understanding of our code.

```
newtype CASStack m a = CASStack (IORef m [a])

push :: MonadConc m => a -> CASStack m a -> m ()
push a (CASStack r) = modifyIORefCAS r (\as -> (a:as, ()))

pop :: MonadConc m => CASStack m a -> m (Maybe a)
pop (CASStack r) = modifyIORefCAS r (\as -> (drop 1 as, listToMaybe as))

peek :: MonadConc m => CASStack m a -> m (Maybe a)
peek (CASStack r) = fmap listToMaybe (readIORef r)
```

Listing 87: A lock-free mutable stack.

```
   popM @  ===  popR @
  peekM @  ===  peekR @
pushM x @  ===  pushR x @
```

Listing 88: Discovering properties between signatures.

Choice of implementation Due to their blocking behaviour, `MVar`s can have poor performance under contention. An alternative concurrency primitive is the `IORef`. An atomic compare-and-swap operation allows threads using `IORef` values to make progress with little overhead, even with contention. Listing 87 shows our implementation, which is similar to the `MVar` stack.

A feature of CoCo that differentiates it from other property-discovery tools is the ability to compare two different signatures which have compatible observation types. We can compare the `MVar` and `IORef` stacks by simply supplying *both signatures* to the tool, each of which contains `push`, `pop`, `peek`, `whenJust`, and `|||`. CoCo then reports 19 properties, including the three in Listing 88. Here we use the list observation again. Functions with names ending `M` are for `MVar` stacks, functions with names ending `R` for `IORef` stacks. These properties tell us what we want to know: the `IORef` stack is equivalent to the `MVar` stack.

A common approach when first writing a program is to do everything in a simple and clearly correct fashion. After checking correctness, we may gradually rewrite components to meet performance requirements. Testing must establish that the rewritten components still exhibit the original behaviour. The ability to determine observational equivalence of different implementations of the same API is an alternative to the more-common unit-testing for this task (He, Hoare, and Sanders 1986).

```
type C = Concurrency

sig :: Sig (QSemN C) Int Int
sig = Sig
  { initialise = new . abs
  , expressions =
    [ lit "wait"   (wait   :: QSemN C -> Int -> C ())
    , lit "signal" (signal :: QSemN C -> Int -> C ())
    ]
  , backgroundExpressions =
    [ commLit "|||" ((|||) :: C A -> C B -> C ())
    , commLit "+"   ((+)   :: Int -> Int -> Int)
    , lit "-"       ((-)   :: Int -> Int -> Int)
    , lit "0" (0 :: Int)
    , lit "1" (1 :: Int)
    ]
  , observe    = \q _ -> remaining q
  , interfere  = \q n -> let i = n `div` 2 in wait q i >> signal q i
  , backToSeed = \q _ -> remaining q
  }
```

Listing 89: A CoCo signature for the `QSemN` type.

7.5.2. Semaphores

A semaphore is a synchronisation primitive used to regulate access to some resource (Dijkstra 1965). A semaphore can be thought of as a record of how many units of some abstract resource are available, with operations to adjust the record in a race-free way. *Binary semaphores* only have two states, and are used to implement locks. *Counting semaphores* have an arbitrary number of states. An implementation of counting semaphores is provided in the Control.Concurrent.QSemN library module. As with the `MVar` and `IORef`, Déjà Fu provides a typeclass-generalised version which we use here.

Listing 89 shows the signature we provide to CoCo. CoCo supports polymorphic function types, as can be seen in the type of `|||`, where `A` and `B` are types we use as *type variables*. The `commLit` function indicates that the supplied binary function is *commutative*, which is used to prune the generated schemas further. The `new`, `wait`, `signal`, and `remaining` functions are provided by the `QSemN` library module. We construct a new semaphore by allocating an arbitrary amount of resource; we observe by checking how much resource remains; and we interfere by taking and then replacing half of the resource. The interference thread is interleaved with the term thread, so it may cause the term thread to block.

CoCo finds 57 properties in this example, so in the remainder of the subsection we

```
(11)          wait @ 0  ===  wait @ 0 >> wait @ 0
(12)        signal @ 0  ===  wait @ 0 >> wait @ 0
(13)        signal @ 1  ===  wait @ (0 - 1)
(14)  signal @ (1 + 1)  ===  wait @ (0 - (1 + 1))
(15)  signal @ (abs x)  ===  wait @ (negate (abs x))
```

Listing 90: Properties about semaphore waiting and signalling.

```
(16)         signal @ 0  ->-  signal @ x >> wait @ x
(17)  signal @ (x + x1)  ->-  signal @ x >> signal @ x1
(18)  signal @ (x + x1)  ->-  (signal @ x) ||| (signal @ x1)
```

Listing 91: Properties suggesting a lack of composability.

only discuss selected properties.

Waiting and signalling CoCo tells us in properties (11) and (12) that the effect of waiting for zero resource and of signalling the availability of zero resource are the same — neither affects the state of the semaphore. Property (11) shows that waiting for zero resource is not a neutral operation, as if it were CoCo would prune the property away. This suggests that `wait` may block.

CoCo also finds properties (13) and (14), revealing another implementation detail, that the programmer can `wait` for a negative value instead of calling `signal`. We might suspect that the more general property `signal @ x === wait @ (-x)` holds for all positive `x`. CoCo finds this form, Property (15), if we extend our signature with `abs` and `negate`.

A lack of composability CoCo reports some strict refinements involving `signal` and `wait`, properties (16–18), where we might expect equivalences. We have just seen with property (15) that funny things happen with negative numbers, so it should be no surprise that these refinements are only equivalences when `x` and `x1` are non-negative.

Types Signalling or awaiting a negative quantity is a breach of the semaphore protocol. Perhaps a better interface for semaphores would only allow nonnegative quantities. The change might avoid accidental breakage in the future if the semantics of negative values are unwittingly changed.

CoCo supports many types, but not all. If the programmer wishes to use types outside of the built-in collection, they must provide some information: a way to enumerate values, an equality predicate, and a symbol to use in variable names. In this way, the

```
       signal @ 0  ===  signal @ n >> wait @ n
signal @ (n + n1)  ===  signal @ n >> signal @ n1
signal @ (n + n1)  ===  (signal @ n) ||| (signal @ n1)
```

Listing 92: Properties (16–18) restricted to natural numbers.

```
-- default, not valid Haskell
readMVar @  ===  readMVar @ >> readMVar @

-- dejafu
check $ sigL (\h0 -> readMVar h0) === sigL (\h0 -> readMVar h0 >> readMVar h0)

-- hunit/tasty
testProperty "name" $
  sigL (\h0 -> readMVar h0) === sigL (\h0 -> readMVar h0 >> readMVar h0)
```

Listing 93: The different CoCo pretty-printing modes.

programmer can extend CoCo to work with arbitrary types, or alter the behaviour of existing types.

If we alter the signature so that `signal` and `wait` use the type of natural numbers rather than integers, properties (16–18) become equivalences, as shown in Listing 92

We could pursue this issue further by examining the terms with Déjà Fu when given a negative quantity, or we could change the type of the function to forbid that case. Ideally, illegal states should be unrepresentable.

7.6. Using CoCo Properties in Déjà Fu

By default, CoCo output is not syntactically valid Haskell. The symbol `@` is not a legal identifier, and the signatures are implicit. So properties cannot simply be copied into a test suite.

Pretty-printing CoCo properties are represented as a pair of expressions and the operator (equality or strict refinement) connecting them. The visual form of the operators is hard-coded, but how the expressions are displayed is controlled by a pretty-printer. The default pretty-printer, used throughout this chapter, favours a concise output. There is also the option to produce Déjà Fu-compatible output, giving properties which can be checked by Déjà Fu directly, or by the hunit-dejafu (Walker 2018a) and tasty-dejafu (Walker 2018b) packages.

Listing 93 shows the three pretty-printing modes. The ‘dejafu’ and ‘hunit/tasty’ modes are valid Haskell, and can be used as a regression test to ensure the property holds (after supplying a signature). These two alternative views of properties are more verbose, and so less convenient to read than the default output when examining a list of properties. Furthermore, it is little work to transform a CoCo property into a form checkable by Déjà Fu.

Signatures Déjà Fu has a notion of signatures, similar to CoCo signatures. A Déjà Fu signature is a simplified form of a CoCo signature: it has an initialisation function, an observation function, an interference function, and a single expression to evaluate. Listing 93 uses signature functions, which take the expression and produce a signature. These functions are called `sigL` because the general form of a CoCo invocation provides two signatures to compare. Discovering properties of a single signature is a special case. So these properties use `sigL`, the 'left signature,' if two signatures were being compared, there would also be reference to a `sigR` (the 'right signature').

CoCo provides a function, `cocoToDejaFu`, to convert a CoCo signature into a Déjà Fu signature function.

Checking properties Déjà Fu can check properties, and also produce a list of counterexamples for failing properties. Properties are evaluated in the same way as CoCo: the behaviours of each term are found with Déjà Fu's systematic concurrency testing functionality, and then these sets of behaviours are compared to check if the property holds. By default, properties are checked with more variable-assignments than CoCo, to increase confidence in the result. The user can also specify the number of seed values and variable-assignments to try, to have even more confidence.

In general it is faster for Déjà Fu to check a property than for CoCo to find it, as to find a property with terms of size n and m, CoCo must first generate and evaluate many expressions smaller than n and m. Déjà Fu just needs to evaluate the two terms in the property.

Like property-testing tools such as QuickCheck (Claessen and Hughes 2000), Déjà Fu uses typeclass polymorphism to enable testing properties which take arguments. If `f x` is testable, and `x` is of some type which can be enumerated by LeanCheck (Braquehais 2017a), then `\x -> f x` is testable.

7.7. Evaluation

Our aim is to help programmers overcome the difficulty of testing concurrent programs. To work towards this, we have presented a new tool, CoCo, to discover behavioural properties of effectful functions operating on shared state.

Applicability beyond Haskell CoCo is tied to Haskell in two ways: it has some knowledge of Haskell types, which is used when generating expressions; and it relies on the Déjà Fu tool to find the results of executing an expression. However, it could be reimplemented for another language. For example, in Erlang the objects of interest are processes. Initialisation is to create a process in a known state. Observation is to send a request for information to a process. Interference is to send messages to a process to change its internal state. The Pulse tool for systematically testing Erlang programs (Claessen, Palka, et al. 2009) would play the part of Déjà Fu.

Value of reported properties Although only supported by a finite number of test cases, the properties reported by CoCo appear accurate in practice. The number of test cases to try is an integer variable in CoCo, so more extensive testing can be performed, at the expense of time and space, by increasing it. These properties can provide helpful insights into the behaviour of functions. As demonstrated in the semaphore case study, surprising properties can suggest that implementations of some functions rely on unstated assumptions. Even without such implementation surprises, it can be difficult to read concurrent source code and grasp all its consequences.

Wide data types CoCo can be made to go wrong. CoCo only performs a limited number of executions for each generated schema, so wide data types pose a problem. In the `MVar` example in Section 7.2, we used `maybe newEmptyMVar newMVar` as our initialisation function. We handle the two cases of `Maybe` differently: in the `Nothing` case, we produce an empty `MVar`; in the `Just` case, we produce a full `MVar`. However, if we had used a type which had more constructors than CoCo performs tests, then some of those constructors would simply never be tested. If each constructor causes some different behaviour, then we will not observe all the behaviours.

Ease of use Ideally, a testing tool should not force the programmer to structure their code in a specific way. CoCo requires the use of the concurrency typeclass used by Déjà Fu, which is not widespread in practice. However, it has been our experience that porting standard Haskell code to the necessary abstraction is a type-directed and mechanical process, requiring little insight.

Interference functions The choice of interference function has a great effect on the discovered properties. Unless two terms are atomic, it is a little suspicious for them to be equivalent. So, to get a fuller understanding of the behaviour of their concurrency functions, the programmer must run CoCo with multiple interference functions. In our experience, interference functions which do one type of interference, such as emptying an `MVar`, tend to give more easily understandable results than interference functions which perform multiple operations.

Automatic interference An alternative to user-supplied interference functions would be for CoCo to generate interference automatically. This is possible, as the primitives (`IORef`s, `MVar`s, and `TVar`s) are all modelled by Déjà Fu. However, it is difficult to see how state with associated invariants can be automatically modified. The user would need to supply predicates to check these invariants, or perhaps a collection of allowable transformations, but is that any better than supplying the interference function?

Term results In its current form, CoCo does not consider term results at all. Only the effects of terms on the distinguished share state are compared. This means that CoCo reports as equivalent terms which could not be substituted for each other at all. It can be useful to see that a complex term has the same effects as a much simpler one, but it would be more useful to know that the more complex one could be replaced by the simpler.

Scaling While a naïve CoCo would scale poorly, our optimisations greatly improve matters in both execution time and maximum resident memory usage. Table 11 on page 148 shows how the three examples we have seen perform with the optimisations disabled. All optimisations are on by default. The most significant improvement by far is **O4**, caching of term behaviours, but the other optimisations all play their part as well. Furthermore, Déjà Fu implements its own optimisations which reduce the time to discover all behaviours of a term.

Optimisations **O1** and **O2** aim to remove uninteresting properties from the CoCo output. Even though **O1** was not motivated by performance, it reduces execution time in the stack and semaphore examples. This is unsurprising, as eliminating terms reduces the amount of evaluation work to be done. Optimisations **O3** and **O4** were motivated by

Term size	1	2	3	4	5	6	7	8
Schemas	15	29	56	88	238	385	1689	2740
Properties	0	0	0	0	1	1	55	55
Time (s)	0.03	0.03	0.45	0.45	9.2	9.2	970	970
Time / schema^2	1.3e-4	3.6e-5	1.4e-4	5.8e-5	1.6e-4	6.2e-5	3.4e-4	1.3e-4

Table 10: Scaling behaviour of the semaphore case study.

```
x == 0  ==>  wait @ x  ===  wait @ x >> wait @ x
```

Listing 94: A property with a precondition.

poor performance, and are clear wins here.

Despite these optimisations, the semaphore example still takes around 15 minutes to run. Table 10 shows how the semaphore case study scales as the term size increases. The execution time grows rapidly, but the time to compare each schema against each other schema, as happens during property discovery, does not. So reducing the number of schemas is the most effective way to reduce the execution time.

One such area for future improvement is in cases where one schema is an instance of another. Such schemas may arise when the signature includes constants. For example, the schema `signal @ 1` is an instance of `signal @ x`. The 'most general term' rule does not apply here, as these are *different* schemas. Constants in signatures are necessary as CoCo does not synthesise preconditions. If it did, constants could be omitted, as any properties which require a specific value for a parameter would be found as a precondition. Property (11) in the semaphore example would instead become the property in Listing 94.

Constants could be removed from the signature, indirectly solving the performance problem. In addition, discovering preconditions would make CoCo able to find properties beyond its current reach (Braquehais and Runciman 2017b).

7.8. Summary

In this chapter we presented CoCo, our tool for automatically discovering properties of functions operating on shared mutable state:

- The properties we discover are equivalences and refinements between the *observable effects* of terms, in the presence of concurrent interference (§7.1).

- We use an approach similar to QuickSpec (Smallbone et al. 2017) and Speculate (Braquehais and Runciman 2017b), but instead consider sets of observations of effects rather than equality of results. We use Déjà Fu to discover these sets (§7.3).
- CoCo properties are conjectures supported only by a finite number of test cases, but appear accurate in practice. However, there are some weaknesses: for example, data types with many constructors are a poor fit for the small number of enumerative tests we perform to decide if a property holds or not. We have not attempted a formal proof of correctness of CoCo (§7.4).
- CoCo is most suited to discovering properties about concurrent data structures. We have demonstrated CoCo with three such examples: `MVar` operations (§7.2), concurrent stacks (§7.5.1), and semaphores (§7.5.2). The semaphore example revealed some surprising details about negative numbers, reinforcing our belief that automatically discovered properties aid program understanding.
- We provide an integration between Déjà Fu and CoCo, so users can take the properties which CoCo discovers and ensure that such properties continue to hold in the future (§7.6).

Property-based testing and property discovery tools have become very popular in the Haskell community. However, these tools work best with pure functions, and only provide limited support for testing effects. Although CoCo is a little rough around the edges, and can be a little confusing to use and interpret, we are helping to extend the reach of property-testing.

Context CoCo does not stand alone, it is related to our other contributions:

- Chapter 5 presents Déjà Fu, the underlying tool which CoCo builds on to test its concurrent expressions. Déjà Fu supports testing CoCo properties, testing with more parameter values than CoCo does, giving a greater degree of confidence.

	Time (s)	Max Residency (kB)	Properties
All On	1.58	3.710	8
O1 Off	1.54	3.710	8
O2 Off	1.57	3.792	9
O3 Off	1.60	3.657	8
O4 Off	26.62	2.114	8
All Off	26.21	8.027	9

(a) The `MVar` example (§7.2).

	Time (s)	Max Residency (MB)	Properties
All On	118	22.30	7
O1 Off	142	25.08	21
O2 Off	111	25.26	12
O3 Off	121	25.04	7
O4 Off	32 341	22.71	7
All Off	54 352	57.31	29

(b) The stack example (§7.5).

	Time (s)	Max Residency (MB)	Properties
All On	945	250.6	55
O1 Off	965	250.6	55
O2 Off	994	261.7	59
O3 Off	983	254.3	55
O4 Off	99 490	242.6	55
All Off	102 759	408.7	59

(c) The semaphore example (§7.5).

O1	is to exclude neutral schemas when generating larger schemas
O2	is to prune properties which are simple consequences of another
O3	is to only evaluate the most general term for each schema
O4	is to cache the behaviours of terms

Table 11: How optimisations alter CoCo's scaling behaviour.

Part III

Conclusions and Future Directions

Chapter 8

Conclusions

We set out to make it easier for programmers to write correct concurrent programs, but have we achieved that? In this chapter we review our contributions and draw some overall conclusions. Recall our motivation from Section 1.2:

> The overall motivation of this research has been to develop tools which make it easier for programmers to write correct concurrent programs. Our primary goal is to demonstrate that concurrency testing techniques, typically described in the context of a simple core language, can be successfully applied to languages with rich concurrency abstractions.

Systematic concurrency testing with rich semantics Chapter 5 introduced Déjà Fu. This is a Haskell tool for testing Haskell programs, but the underlying techniques are not Haskell specific. Haskell has an unusually rich concurrency abstraction, whereas SCT techniques are typically described in the literature for simple concurrency abstractions. Even real programming languages tend to have simple concurrency abstractions. Maple (Yu et al. 2012) is able to test arbitrary pthread programs by considering just 19 primitive actions, whereas the expression of Haskell concurrency in Déjà Fu requires 34 just for concurrency, and a further 9 for STM. The number of primitive actions a concurrency testing tool must consider is only an indirect measure of the complexity of the concurrency model it supports, but such a large difference is suggestive.

In Haskell, there are many different operations with partially overlapping behaviour. It is not clear that a typical SCT algorithm would work effectively in this context. Our case studies in Section 5.9 provide a convincing demonstration that SCT can be applied to languages with rich concurrency abstractions.

Effective bug finding with randomised scheduling Chapter 6 introduced the swarm scheduling algorithm for random testing of concurrent programs. Random testing is useful because it can be both fast and effective, even in cases where complete testing does not scale. Our benchmark results in Section 6.3 show that it performs as well as the PCT algorithm (Burckhardt et al. 2010) in terms of bug-finding ability. Crucially, PCT requires the user to supply parameters derived from the program under test, whereas swarm scheduling does not. The freedom from any such requirement makes swarm scheduling simpler to implement and use than PCT, yet it still finds bugs just as well.

Discovering properties of concurrent programs Chapter 7 introduced CoCo. By synthesising program terms and performing property-based testing, CoCo can give the programmer new insights into their code. Like Déjà Fu, this is a Haskell tool, but the techniques are not Haskell specific. The underlying idea is that we can compare sets of program behaviours to make meaningful claims about the relation between the components which make up those programs. Our case studies in Section 7.5 show the sorts of properties we can discover but, as we see in Section 7.7, CoCo has scaling difficulties.

The CoCo approach applies not just to concurrent programs, but to nondeterministic programs in general. If we have an efficient, but nondeterministic, algorithm for a problem, we may wish to be able to use it in place of a slow, but deterministic, algorithm. The deterministic algorithm is a refinement of the nondeterministic algorithm, which may introduce additional behaviours.

Drawbacks of refactoring A weakness of our Haskell work is the `MonadConc` typeclass. Requiring programmers to modify their code, even in a straightforward way, is a barrier to entry that many will not wish to overcome. Furthermore, when typeclass-polymorphic code is compiled, the definitions of typeclass member functions cannot be inlined, as they are not known (Peyton Jones and Marlow 2002). The recent Backpack work (E. Z. Yang 2017) offers an alternative here, lessening the code modification problem and solving the optimisation problem.

The inevitable exponentials Concurrent programs are nondeterministic, and this is where the difficulty of writing correct concurrent programs comes from. Testing a concurrent program requires executing it multiple times with different schedules. This multiplicity adds overhead compared to sequential tests, where a single execution suffices. Even worse, a concurrent program with n threads which each execute for at most k steps can have as many as $\frac{(nk)!}{(k!)^n}$ executions (Madanlal Musuvathi and Qadeer 2007)!

Déjà Fu implements schedule bounding (Emmi, Qadeer, and Rakamarić 2011; Madanlal Musuvathi and Qadeer 2008; Madanlal Musuvathi and Qadeer 2007) and partial-order reduction (Flanagan and Godefroid 2005; Godefroid 1996) to improve the average case, but the worst case remains a possibility. Empirical studies show that small test cases with just two threads and two pre-emptive context switches suffice for finding many real-world concurrency bugs (Thomson, Donaldson, and Betts 2014). There is a *small-scope hypothesis* here: most concurrency bugs do not only arise in complicated test cases; rather, we just need a handful of actions to happen in the wrong order. This is the intuition behind PCT (Burckhardt et al. 2010). So there is a terrible asymptotic worst case, but in practice test cases are often small. When test cases *are* too large for systematic testing to effectively explore the state-space, then we can use a random approach, as we did in Chapter 6.

The difficulty of interpreting success It can be difficult to look at the result of a successful concurrency test and know what it is telling us. We saw this with CoCo in Section 7.2, where the programmer may need to run the tool with a variety of interference functions to see the full picture. Properties found with one sort of concurrent interference may not generalise to cases with different interference. Similarly, successful concurrency tests in Déjà Fu may not generalise to cases where the concurrent environment is different. This difficulty is related to the problem of judging the quality of a test suite, which we will discuss in Chapter 9.

The difficulty of interpreting failure It can be difficult to look at a failing concurrency test and diagnose the problem. We saw this with Déjà Fu in Section 5.9.1, where the resulting execution traces were large and difficult to follow. Traces are a low-level construct: they may become invalid when library dependencies change, even if the key scheduling decisions remain the same. Which information is truly important? It is not obvious.

Overall conclusions Concurrency errors, sometimes called "Heisenbugs" due to their unpredictable behaviour, can be among the most difficult to debug (Madanlal Musuvathi, Qadeer, et al. 2008). The ideas behind concurrency testing have been around for some time now (Godefroid 1996), and yet concurrency testing tools are not widely used. By contributing new tools and illustrating what they can do we hope to help address this problem.

Chapter 9

Future Directions

We have made our contributions, and even found some users. But Déjà Fu, and concurrency testing in general, is not done yet. We now discuss future directions for the tools we have developed, and end with a hopeful vision for concurrency testing in the future.

Measuring the quality of test suites How do we come to believe that a test suite is strong evidence for the correctness of some program? Any testing regimen is only as good as its tests. For sequential programs, we can use the traditional metric of code coverage. Code which is not covered at all usually has more bugs than code which is covered by even low-quality tests (Ahmed et al. 2016). For concurrent programs, what metric do we use? If it is some notion of coverage, what is the space being covered? Here are two candidates:

- *Schedule-sensitive branches* are often unintentional and erroneous points of synchronisation between concurrent threads (J. Huang and Rauchwerger 2015). A good concurrency test suite should try all cases in a schedule-sensitive branch.
- *Unguarded shared state* without a synchronisation mechanism can lead to invalid or corrupt data. If we have functions which operate on some mutable state of the same type, then a good concurrency test suite should check what happens when that state is shared and the functions are executed concurrently.

Both statement coverage and mutation score have only a weak negative correlation with bug fixes (Ahmed et al. 2016), but there *is* a statistically significant difference between uncovered code and code with some, even if low, coverage (Ahmed et al. 2016). Being able to identify the uncovered gaps of a concurrency test suite could greatly help with improving the overall quality of a piece of software.

Maximal causality reduction for Déjà Fu The MCR algorithm (J. Huang 2015) explores a provably minimal number of schedules required for completeness. Typically this is orders of magnitude fewer than the number of schedules constrained only by dynamic partial-order reduction. However, MCR is tricky to implement in Haskell as it requires local determinism: the future actions of a thread are determined solely by the prior actions of the same thread and shared variables it has read. Haskell breaks local determinism with asynchronous exceptions, where one thread can kill another.

It may be possible to implement a Haskell-MCR by translating Haskell execution traces into a simpler form suitable for MCR. For example, asynchronous exceptions can be modelled by giving each thread an exception variable: throwing an exception to a thread writes to its exception variable, and the thread checks its exception variable before each action. This polling technique is similar to how an operating system can abstract over hardware interrupts: when an interrupt arrives, its exception handler sets a flag and returns control to the interrupted routine, which checks the flag at a convenient point.

Optimal DPOR for Déjà Fu The Optimal DPOR with Observers algorithm (Aronis et al. 2018) is based on the same insight as MCR, that the order of writes to some shared state only matters if there exists a read operation which observes the difference. As it is a DPOR algorithm, it could be closer to what Déjà Fu currently does, and simpler to implement. However, it is even more restrictive than MCR's requirement of local determinism, forbidding threads from disabling each other entirely. This means that blocking operations currently cannot be modelled with the Optimal DPOR with Observers algorithm, so it is currently unsuitable for Déjà Fu.

Accurately modelling delays in Déjà Fu Some users have expressed interest in using Déjà Fu to test systems where accurate timing is important[1], such as distributed systems with timeouts. However, Déjà Fu currently has no notion of time. A thread delaying is treated just the same as a thread yielding. It has no further effect on how threads are scheduled during testing.

Listing 95 shows an example of a program with a delay: an `IORef` is created holding the value `False`, which is set to `True` by another thread after a one hour delay. Immediately after forking the second thread, the `IORef` is read and its value returned. What should a timing-aware Déjà Fu say about this program? Currently, both `True` and `False` are reported as possible outcomes. In reality, however, getting `True` requires the main

[1] https://github.com/barrucadu/dejafu/issues/130

```haskell
example :: MonadConc m => m Bool
example = do
    r <- newIORef False
    fork (threadDelay 1000000 >> writeIORef r True)
    readIORef r
```

Listing 95: A program with a large delay.

thread to not be scheduled for the entire one hour delay, which is vanishingly unlikely. So should the `True` case be forbidden? When there are multiple threads with delays which are important relative to each other, the problem only becomes more confusing.

Invariant testing in Déjà Fu Temporal program logics (Pnueli 1977) allow defining and checking complex specifications for how a program should behave over the course of its entire execution. While Déjà Fu is not a temporal logic checking tool, it can get part of the way there by allowing users to specify invariants which are checked continuously while the program under test runs. As Déjà Fu drives the execution, these invariants can be checked atomically. This would be similar to, but more general than, the GHC Haskell function `always :: STM Bool -> STM ()`, which registers an invariant to be checked at the end of STM transactions.

Automatic interference for CoCo CoCo requires the user to provide the interference functions which are necessary to distinguish between atomic and non-atomic actions. An alternative would be for CoCo to generate interference automatically. This is possible, as the primitives (`IORef`s, `MVar`s, and `TVar`s) are all modelled by Déjà Fu. However, it is difficult to see how state with associated invariants can be automatically modified. The user would need to supply predicates to check these invariants, or perhaps a collection of allowable transformations, but is that any better than supplying the interference function directly?

Conditional properties in CoCo Speculate (Braquehais and Runciman 2017b) discovers conditional equations and inequalities automatically, which greatly expands the range of properties which can be found. Conditional properties are useful as we see how our functions behave in different situations, rather than just in general. CoCo has a limited form of conditional properties, involving preconditions on the generated seed values. Such precondition functions must be supplied by the user. However, it would be much more useful if CoCo could synthesise preconditions, as Speculate does, to discover interesting cases itself.

Term rewriting for CoCo Both QuickSpec (Smallbone et al. 2017) and Speculate (Braquehais and Runciman 2017b) use term rewriting to prune the discovered properties and to avoid testing many cases. Pruning by reducing to a normal form is difficult to do with concurrency, as effects may be non-local. For example, consider with relaxed memory where writes to shared variables may be delayed (Zhang, Kusano, and Wang 2015). Such behaviours make the effect of composing two terms far less predictable. Even so, it may still be possible in some cases to use something like term rewriting to prune properties.

The future Testing a concurrent program goes something like this: (1) write a small concurrent program; (2) run it lots of times with your concurrency testing tool, recording the results of each execution; (3) look at the collection of results. This approach is rather different to how we test sequential programs.

Test cases for sequential programs are generally written in a three-part "given, when, then" style (Fowler 2013). The "given" sets up the system under test, the "when" exercises it in some way, and the "then" decides if the test passes or fails. But in a test case for a concurrent program, we may never reach the "then". If a test case has some concurrency failure, such as deadlock, we do not see the full picture. Did the program only deadlock, or did it also corrupt its state? There is a fundamental difference between normal program errors and concurrency errors.

Things do not need to be this way. By controlling the concurrency, a testing tool can ensure that even if the "when" component enters a failure state, the "then" component is still executed. When a failure is reported, a *simplified* description of the execution of the "when" component, containing just the key details, can be given to the user. Maybe this is not a complete trace: the Concurrit (Elmas et al. 2013) tool offers an alternative approach, where executions are represented by a small collection of scheduling constraints.

However, testing concurrent programs will always remain a little more difficult than testing sequential programs. Concurrency bugs are fundamentally more complex than other bugs, and there is only so much that tooling and abstraction can accomplish.

Appendix A

Haskell Reference

This appendix gives a brief introduction to Haskell syntax. We assume familiarity with functional programming, in particular: with currying, first-class functions, pattern matching, and parametric polymorphism. For a more gentle introduction to Haskell, see the tutorials page on the Haskell wiki[1].

- `foo = bar` binds a name to a value

  ```
  five = 5
  ```

- Function definitions add argument names after the function name

  ```
  increment n = n + 1
  ```

- Function calls have no parentheses

  ```
  six = increment five
  ```

- Function calls are left associative

  ```
  seven = increment (increment five)
  ```

- Function calls take precendence over operators

  ```
  incAndAdd x y = increment x + increment y
  ```

- Anonymous functions are declared with `\args -> expr`

  ```
  double = \x -> incAndAdd x x - 2
  ```

- Functions are curried, 'multi-argument functions' are syntactic sugar

  ```
  incAndAdd x = \y -> increment x + increment y
  ```

[1] https://wiki.haskell.org/Tutorials

- Functions are first-class values

```
compose f g x = f (g x)
```

- Operators are just functions with symbolic names

```
f . g = compose f g
```

- The $ operator is right-associative function application

```
compose f g x = f $ g x
```

- We can use a `let` expression to introduce local definitions

```
sumOf3 x y z = let temp = x + y in temp + z
```

- Or a `where` clause

```
sumOf3 x y z = temp + z where
  temp = x + y
```

- `type` gives an existing type a new alias

```
type Name = String
```

- We can annotate a definition or a value with a type using `::`

```
name :: Name
name = "Michael Walker"
```

- `data` defines a new data type

```
data Colour
  = RGB Int Int Int
  | Grey Double

magenta :: Colour
magenta = RGB 255 0 255
```

- Records allow us to name the fields in a type

```
data RGB = RGB
  { red   :: Int
  , green :: Int
  , blue  :: Int
  }

red :: RGB
red = RGB { red = 255, green = 0, blue = 0 }
```

- -> denotes the type of a function

  ```
  sum3 :: Int -> Int -> Int -> Int
  sum3 x y z = x + y + z
  ```

- -> is right associative, these types are all equivalent

  ```
  sum3 :: Int -> Int -> Int -> Int
  sum3 :: Int -> Int -> (Int -> Int)
  sum3 :: Int -> (Int -> Int -> Int)
  sum3 :: Int -> (Int -> (Int -> Int))
  ```

- Names that start with an upper case letter in types are concrete types

  ```
  intId :: Int -> Int
  intId x = x
  ```

- Names that start with a lower case letter in types are type variables

  ```
  id :: a -> a
  id x = x
  ```

- We can use type variables when defining types

  ```
  data Maybe  a   = Just a | Nothing
  data Either a b = Left a | Right b
  ```

- List comprehensions give a concise syntax to construct new lists

  ```
  [(a, b) | a <- [1,2], b <- ['x','y']] == [(1,'x'), (1,'y'), (2, 'x'), (2, 'y')]
  ```

- List comprehensions can contain guards

  ```
  [(a, b) | a <- [1,2], b <- ['x','y'], a == 2] == [(2, 'x'), (2, 'y')]
  ```

- The list type constructors are `:` and `[]`

  ```
  -- sugar: not really a legal data definition!
  data [a] = a : [a] | []
  ```

- There are also tuples, each size a different type

  ```
  -- sugar: not really legal data definitions!
  data (a, b) = (a, b)
  data (a, b, c) = (a, b, c)
  -- ...
  ```

- The ‘unit’ type looks like an empty tuple

  ```
  -- sugar: not really a legal data definition!
  data () = ()
  ```

- `case` expressions let us pattern match on data constructors

```
fromMaybe :: a -> Maybe a -> a
fromMaybe def m = case m of
  Just a  -> a
  Nothing -> def
```

- Function definitions can also pattern match on their arguments

```
maybe :: a -> (a -> b) -> Maybe a -> b
maybe def f (Just a) = f a
maybe def f Nothing  = def
```

- The pattern _ matches anything

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True t _ = t
ifThenElse _    _ f = f
```

- Typeclasses are used to group common behaviour, such as equality

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  True  == True  = True
  False == False = True
  _ == _ = False
```

- Typeclass constraints can appear in type signatures

```
eq3 :: Eq a => a -> a -> a -> Bool
eq3 a b c = a == b && b == c
```

- The `Functor` typeclass represents 'contexts' which can be mapped over

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- `fmap` is sometimes written <$>

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

- The `Applicative` typeclass extends `Functor` with the ability to wrap up a value

```
class Functor f => Applicative f where
  pure :: a -> f a
```

- ...and to apply a value-in-a-context to a function-in-a-context

  ```
  class Functor f => Applicative f where
    (<*>) :: f (a -> b) -> f a -> f b
    pure  :: a -> f a
  ```

- The `Monad` typeclass extends `Applicative` with the ability to compose contexts

  ```
  class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: Monad m => m a -> m b -> m b
  ma >> mb = ma >>= \_ -> mb
  ```

- do-notation is a syntactic sugar for sequencing monadic actions

  ```
  main :: IO ()
  main = do
    putStrLn "hello"
    putStrLn "world"
  ```

- <- is used to bind the result of a monadic action to a name in do-notation

  ```
  main :: IO ()
  main = do
    putStrLn "What's your name?"
    name <- getLine
    putStrLn ("Hello " ++ name)
  ```

- `let` is used to bind an expression to a name in do-notation

  ```
  main :: IO ()
  main = do
    let sum = sum3 1 10 100
    print sum
  ```

Language extensions GHC provides a rich collection of extensions to standard Haskell. These are documented fully in the GHC manual[2].

- The `ViewPatterns` language extension allows pattern matching on a function result

  ```
  myFst :: (a, b) -> a
  myFst (fst -> a) = a
  ```

[2] http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/lang.html

- The `RankNTypes` language extension allows universally quantified type variables

  ```
  pair :: (forall x. x -> x) -> a -> b -> (a, b)
  pair f a b = (f a, f b)
  ```

- The `TypeFamilies` language extension allows types to be associated with a type-class

  ```
  class Tower a where
    type Next a :: *
    promote :: a -> Next a

  instance Tower Word8 where
    type Next Word8 = Word16
    promote = fromIntegral
  ```

Appendix B

Swarm Scheduling Algorithm

Listing 96 shows the core of the swarm scheduling algorithm from Chapter 6, as implemented in C++ in Maple. There are four methods:

- `Explore` is called by Maple, and is what drives the execution of the program. It picks a thread to run and calls `Execute`, which causes Maple to advance that thread to the next scheduling point.
- `AssignWeightsToNewThreads` is called by `Explore`. It checks that every enabled thread has a weight in the `weights` map, assigning a weight if not. We only consider enabled threads, so if a thread is blocked when it is created and never becomes enabled, it will never receive a weight.
- `AssignWeightTo` is called by `AssignWeightsToNewThreads` for each new thread, and by `Explore` when a weight change point is encountered. It assigns a new weight in the range $[w_{min}, w_{max}]$, which we have hard-coded here to be 1 and 50, respectively.
- `PickNextRandom` is called by `Explore`, and is what chooses the thread. It produces a list of the weights of all enabled threads, and uses that list as a discrete distribution. As this method is called after `AssignWeightsToNewThreads`, every enabled thread will have a weight.

```cpp
void SwarmScheduler::Explore(State *init_state) {
  State *state = init_state;
  unsigned int steps = 0;

  while (!state->IsTerminal()) {
    AssignWeightsToNewThreads(state);

    auto it = PickNextRandom(state);
    for(unsigned int cpoint : changePoints) {
      if(steps == cpoint) {
        AssignWeightTo(it.first->uid());
      }
    }

    state = Execute(state, it.second);
    steps++;
  }
}

void SwarmScheduler::AssignWeightsToNewThreads(State *state) {
  auto enabled = state->enabled();
  for(auto it = enabled->begin(); it != enabled->end(); it++) {
    auto threadId = it->first->uid();
    if(weights.find(threadId) == weights.end()) {
      AssignWeightTo(threadId);
    }
  }
}

void SwarmScheduler::AssignWeightTo(uint32 threadId) {
  std::uniform_int_distribution<uint8> weightDist(1, 50);
  weights[threadId] = weightDist(random);
}

std::pair<systematic::Thread* const, systematic::Action*>
SwarmScheduler::PickNextRandom(State *state) {
  auto enabled = state->enabled();

  // get the enabled threads and their weights
  std::list<uint8> e_ws;
  for(auto it = enabled->begin(); it != enabled->end(); it++) {
    e_ws.push_back(weights[it->first->uid()]);
  }

  // pick an enabled thread by a weighted random choice
  auto it = enabled->begin();
  std::discrete_distribution<int> dist(e_ws.begin(), e_ws.end());
  std::advance(it, dist(random));
  return *it;
}
```

Listing 96: The core of the C++ swarm scheduling algorithm, implemented in Maple.

Bibliography

Abdelrasoul, Mahmoud (2017). "Promoting Secondary Orders of Event Pairs in Randomized Scheduling Using a Randomized Stride". In: *Proceedings of the 32nd IEEE / ACM International Conference on Automated Software Engineering*. ASE 2017. IEEE Press, pp. 741–752. DOI: 10.1109/ASE.2017.8115685.

Ahmed, Iftekhar et al. (2016). "Can Testedness Be Effectively Measured?" In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. ACM, pp. 547–558. DOI: 10.1145/2950290.2950324.

Anisimov, A. V. and D. E. Knuth (1979). "Inhomogeneous sorting". In: *International Journal of Computer & Information Sciences* 8.4, pp. 255–260. DOI: 10.1007/BF00993053.

Ankuzik (2014). "Haskell, Testing a Multithreaded Application". At http://kukuruku.co/hub/haskell/haskell-testing-a-multithread-application, accessed at 2018-01-13.

Aronis, Stavros et al. (2018). "Optimal Dynamic Partial Order Reduction with Observers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pp. 229–248. DOI: 10.1007/978-3-319-89963-3_14.

Arts, Thomas, John Hughes, Joakim Johansson, et al. (2006). "Testing Telecoms Software with Quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. ERLANG '06. ACM, pp. 2–10. DOI: 10.1145/1159789.1159792.

Arts, Thomas, John Hughes, Ulf Norell, et al. (2011). "Accelerating Race Condition Detection Through Procrastination".

In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. Erlang '11. ACM, pp. 14–22. DOI: 10.1145/2034654.2034659.

Bolingbroke, Max (2017). "test-framework: Framework for running and organising tests, with HUnit and QuickCheck support". At https://hackage.haskell.org/package/test-framework, accessed at 2018-01-25.

Böttcher, Thomas and Frank Huch (2002). "A Debugger for Concurrent Haskell". In: *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*. IFL '02.

Braquehais, Rudy (2017a). "leancheck: Cholesterol-free property-based testing". At https://hackage.haskell.org/package/leancheck, accessed at 2018-01-13.

– (2017b). "Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing". PhD thesis.

Braquehais, Rudy and Colin Runciman (2017a). "Extrapolate: generalizing counter-examples of functional test properties". In: *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '17.

– (2017b). "Speculate: Discovering Conditional Equations and Inequalities About Black-box Functions by Reasoning from Test Results". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. ACM, pp. 40–51. DOI: 10.1145/3122955.3122961.

Burckhardt, Sebastian et al. (2010). "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. ACM, pp. 167–178. DOI: 10.1145/1735971.1736040.

Burnim, Jacob and Koushik Sen (2010). "DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs". In: *Proceedings of the 32nd ACM / IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. ACM, pp. 415–424. DOI: 10.1145/1806799.1806860.

Cheplyaka, Roman (2018). "tasty: Modern and extensible testing framework". At https://hackage.haskell.org/package/tasty, accessed at 2018-01-25.

Christakis, Maria, Alkis Gotovos, and Konstantinos Sagonas (2013). "Systematic Testing for Detecting Concurrency Errors in Erlang Programs". In: *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing,* *Verification and Validation*. ICST '13. IEEE Computer Society, pp. 154–163. DOI: 10.1109/ICST.2013.50.

Claessen, Koen and John Hughes (2000). "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP'00. ACM, pp. 268–279. DOI: 10.1145/351240.351266.

Claessen, Koen, Michal Palka, et al. (2009). "Finding race conditions in Erlang with QuickCheck and PULSE". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. ACM, pp. 149–160. DOI: 10.1145/1596550.1596574.

Claessen, Koen, Nicholas Smallbone, and John Hughes (2010). "QuickSpec: Guessing Formal Specifications Using Testing". In: *Proceedings of the 4th International Conference on Tests and Proofs*. TAP'10. Springer-Verlag, pp. 6–21. DOI: 10.1007/978-3-642-13977-2_3.

Coons, Katherine E., Madan Musuvathi, and Kathryn S. McKinley (2013). "Bounded Partial-order Reduction". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '13. ACM, pp. 833–848. DOI: 10.1145/2544173.2509556.

Cordeiro, Lucas and Bernd Fischer (2011). "Verifying Multi-threaded Software Using Smt-based Context-bounded Model Checking". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. ACM, pp. 331–340. DOI: 10.1145/1985793.1985839.

De Moura, Leonardo and Nikolaj Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08 / ETAPS'08. Springer-Verlag, pp. 337–340.

– (2011). "Satisfiability Modulo Theories: Introduction and Applications". In: *Commun. ACM* 54.9, pp. 69–77. DOI: 10.1145/1995376.1995394.

Dice, David, Danny Hendler, and Ilya Mirsky (2013). "Lightweight Contention Management for Efficient Compare-and-swap Operations". In: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Springer-Verlag, pp. 595–606. DOI: 10.1007/978-3-642-40047-6_60.

Dijkstra, Edsger W. (1965). "Cooperating sequential processes". Published as EWD123.

Dodds, Mike, Andreas Haas, and Christoph M. Kirsch (2015). "A Scalable, Correct Time-Stamped Stack". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. ACM, pp. 233–246. DOI: 10.1145/2676726.2676963.

Eich, Brendan and Mozilla Foundation (1996). "Mozilla SpiderMonkey". At https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, accessed at 2018-01-25.

Elmas, Tayfun et al. (2013). "CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. ACM, pp. 153–164. DOI: 10.1145/2491956.2462162.

Emmi, Michael, Shaz Qadeer, and Zvonimir Rakamarić (2011). "Delay-bounded Scheduling". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. ACM, pp. 411–422. DOI: 10.1145/1925844.1926432.

Ernst, Michael D. et al. (2007). "The Daikon System for Dynamic Detection of Likely Invariants". In: *Sci. Comput. Program.* 69.1-3, pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

Flanagan, Cormac and Patrice Godefroid (2005). "Dynamic partial-order reduction for model checking software". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. ACM, pp. 110–121. DOI: 10.1145/1040305.1040315.

Fowler, Martin (2013). "GivenWhenThen". At https://martinfowler.com/bliki/GivenWhenThen.html, accessed at 2018-02-19.

Godefroid, Patrice (1996). "Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem". PhD thesis. DOI: 10.1007/3-540-60761-7.

Godefroid, Patrice and Didier Pirottin (1993). "Refining Dependencies Improves Partial-Order Verification Methods (extended abstract)". In: *Computer Aided Verification: 5th International Conference, CAV '93 Elounda, Greece,* *June 28–July 1, 1993 Proceedings*. Ed. by Costas Courcoubetis. Springer Berlin Heidelberg, pp. 438–449. DOI: 10.1007/3-540-56922-7_36.

Groce, Alex et al. (2012). "Swarm Testing". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. ACM, pp. 78–88. DOI: 10.1145/2338965.2336763.

Harris, Tim et al. (2005). "Composable Memory Transactions". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. ACM, pp. 48–60. DOI: 10.1145/1065944.1065952.

He, J., C. A. R. Hoare, and J. W. Sanders (1986). "Data refinement refined resume". In: *ESOP 86: European Symposium on Programming Saarbrücken, Federal Republic of Germany March 17–19, 1986 Proceedings*. Springer Berlin Heidelberg, pp. 187–196.

Herington, Dean and Simon Hengel (2017). "HUnit: A unit testing framework for Haskell". At https://hackage.haskell.org/package/HUnit, accessed at 2018-01-25.

Holser, Paul (2018). "junit-quickcheck: Property-based testing, JUnit-style". At https://github.com/pholser/junit-quickcheck, accessed at 2018-01-22.

Huang, Jeff (2015). "Stateless Model Checking Concurrent Programs with Maximal Causality Reduction". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, pp. 165–174. DOI: 10.1145/2737924.2737975.

Huang, Jeff and Lawrence Rauchwerger (2015). "Finding Schedule-sensitive Branches". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. ACM, pp. 439–449. DOI: 10.1145/2786805.2786840.

Huang, Shiyou and Jeff Huang (2016). "Maximal Causality Reduction for TSO and PSO". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. ACM, pp. 447–461. DOI: 10.1145/2983990.2984025.

– (2017). "Speeding Up Maximal Causality Reduction with Static Dependency Analysis". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Ed. by Peter Müller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs).

Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 16:1–16:22. DOI: 10.4230/LIPIcs.ECOOP.2017.16.

HypothesisWorks (2018). "hypothesis-python: Advanced property-based (QuickCheck-like) testing for Python". At https://github.com/HypothesisWorks/hypothesis-python, accessed at 2018-01-22.

IEEE (1995). *POSIX.1c, Threads extensions (IEEE Standard 1003.1c-1995).* IEEE.

Kusano, Markus, Arijit Chattopadhyay, and Chao Wang (2015). "Dynamic Generation of Likely Invariants for Multithreaded Programs". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1.* ICSE '15. IEEE Press, pp. 835–846. DOI: 10.1109/ICSE.2015.95.

Lamport, L. (1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Trans. Comput.* 28.9, pp. 690–691. DOI: 10.1109/TC.1979.1675439.

Le, Matthew and Matthew Fluet (2015). "Partial Aborts for Transactions via First-class Continuations". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming.* ICFP 2015. ACM, pp. 230–242. DOI: 10.1145/2784731.2784736.

Lea, Doug (1996). *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc.

Marlow, Simon (2013). *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming.* O'Reilly Media.

– (2017). "async: Run IO operations asynchronously and wait for their results". At https://hackage.haskell.org/package/atomic-primops, accessed at 2018-01-25.

Marlow, Simon and Ryan R. Newton (2016). "monad-par: A library for parallel programming based on a monad". At https://hackage.haskell.org/package/monad-par, accessed at 2018-02-08.

Marlow, Simon, Ryan Newton, and Simon Peyton Jones (2011). "A Monad for Deterministic Parallelism". In: *Proceedings of the 4th ACM Symposium on Haskell.* Haskell '11. ACM, pp. 71–82. DOI: 10.1145/2034675.2034685.

Marlow, Simon, Simon Peyton Jones, and Satnam Singh (2009). "Runtime Support for Multicore Haskell". In: *Proceedings of the 14th ACM*

SIGPLAN International Conference on Functional Programming. ICFP '09. ACM, pp. 65–78. DOI: 10.1145/1596550.1596563.

Mazurkiewicz, Antoni (1986). "Trace theory". In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer Berlin Heidelberg, pp. 278–324.

McPherson, Cal (2004). "Haskell, Testing a Multithreaded Application". At http://ctrace.sourceforge.net/, accessed at 2018-01-25.

Mozilla Foundation (1996). "Netscape Portable Runtime". At https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR, accessed at 2018-01-25.

Musuvathi, Madan and Shaz Qadeer (2007). "CHESS: Systematic Stress Testing of Concurrent Software". In: *Proceedings of the 16th International Conference on Logic-based Program Synthesis and Transformation*. LOPSTR'06. Springer-Verlag, pp. 15–16. DOI: 10.1007/978-3-540-71410-1_2.

Musuvathi, Madanlal and Shaz Qadeer (2007). "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. ACM, pp. 446–455. DOI: 10.1145/1273442.1250785.

– (2008). "Fair Stateless Model Checking". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. ACM, pp. 362–371. DOI: 10.1145/1375581.1375625.

Musuvathi, Madanlal, Shaz Qadeer, et al. (2008). "Finding and Reproducing Heisenbugs in Concurrent Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. USENIX Association, pp. 267–280.

Newton, Ryan R. (2017). "atomic-primops: A safe approach to CAS and other atomic ops in Haskell". At https://hackage.haskell.org/package/atomic-primops, accessed at 2018-01-25.

Oracle (2017). "Java Thread Primitive Deprecation". At https://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html, accessed at 2018-01-13.

Owens, Scott, Susmit Sarkar, and Peter Sewell (2009). "A Better x86 Memory Model: X86-TSO". In: *Proceedings of the 22nd International*

Conference on Theorem Proving in Higher Order Logics. TPHOLs '09. Springer-Verlag, pp. 391–407. DOI: 10.1007/978-3-642-03359-9_27.

Peyton Jones, Simon (2002). "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell". In:

Peyton Jones, Simon, Andrew Gordon, and Sigbjorn Finne (1996). "Concurrent Haskell". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. ACM, pp. 295–308. DOI: 10.1145/237721.237794.

Peyton Jones, Simon and Simon Marlow (2002). "Secrets of the Glasgow Haskell Compiler Inliner". In: *J. Funct. Program.* 12.5, pp. 393–434. DOI: 10.1017/S0956796802004331.

Pike, Lee (2014). "SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. ACM, pp. 53–64. DOI: 10.1145/2633357.2633365.

Pnueli, Amir (1977). "The Temporal Logic of Programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. IEEE Computer Society, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

Pratchett, Terry (2001). *Thief of Time*. HarperCollins.

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad (2008). "SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell'08. ACM, pp. 37–48. DOI: 10.1145/1411286.1411292.

Russell, Stuart and Peter Norvig (2002). *Artificial Intelligence: A Modern Approach*. 2nd. Prentice Hall Press.

Sen, Koushik (2007). "Effective Random Testing of Concurrent Programs". In: *Proceedings of the Twenty-second IEEE / ACM International Conference on Automated Software Engineering*. ASE '07. ACM, pp. 323–332. DOI: 10.1145/1321631.1321679.

– (2008). "Race Directed Random Testing of Concurrent Programs". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. ACM, pp. 11–21. DOI: 10.1145/1375581.1375584.

Serebryany, Konstantin and Timur Iskhodzhanov (2009). "ThreadSanitizer: Data Race Detection in Practice".

In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. ACM, pp. 62–71. DOI: 10.1145/1791194.1791203.

Shavit, Nir and Dan Touitou (1995). "Software Transactional Memory". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. ACM, pp. 204–213. DOI: 10.1145/224964.224987.

Siberschatz, Abraham and Peter B. Galvin (1993). *Operating System Concepts, 4th Ed.* 4th. Addison-Wesley Longman Publishing Co., Inc.

Smallbone, Nicholas et al. (2017). "Quick specifications for the busy programmer". In: *Journal of Functional Programming* 27. DOI: 10.1017/S0956796817000090.

Smith, Calvin, Gabriel Ferns, and Aws Albarghouthi (2017). "Discovering Relational Specifications". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. ACM, pp. 616–626. DOI: 10.1145/3106237.3106279.

Snoyman, Michael (2016). "auto-update: Efficiently run periodic, on-demand actions". At https://hackage.haskell.org/package/auto-update, accessed at 2018-01-25.

SPARC International Inc., CORPORATE (1992). *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc.

Stanley, Jacob (2018). "leancheck: Cholesterol-free property-based testing". At https://hackage.haskell.org/package/hedgehog, accessed at 2018-03-10.

The GHC Developers (2017a). "Control.Concurrent module documentation". At https://hackage.haskell.org/package/base-4.10.0.0/docs/Control-Concurrent.html, accessed at 2018-01-25.

– (2017b). "Data.IORef module documentation". At https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-IORef.html, accessed at 2018-01-13.

The Go Developers (2017a). "Data Race Detector". At https://golang.org/doc/articles/race_detector.html, accessed at 2018-01-13.

– (2017b). "testing/quick package documentation". At https://golang.org/pkg/testing/quick/, accessed at 2018-01-22.

The JUnit Developers (2017). "Parameterized tests documentation". At https://github.com/junit-team/junit4/wiki/Parameterized-tests, accessed at 2018-01-22.

The NUnit Developers (2017a). "RandomAttribute documentation". At https://github.com/nunit/docs/wiki/Random-Attribute, accessed at 2018-01-22.

– (2017b). "ValuesAttribute documentation". At https://github.com/nunit/docs/wiki/Values-Attribute, accessed at 2018-01-22.

The Rust Developers (2011). *The Rust Programming Language (first edition).*

– (2018). "catch_unwind function documentation". At https://doc.rust-lang.org/1.23.0/std/panic/fn.catch_unwind.html, accessed at 2018-01-13.

Thomson, Paul, Alastair F. Donaldson, and Adam Betts (2014). "Concurrency Testing Using Schedule Bounding: an Empirical Study". In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, pp. 15–28. DOI: 10.1145/2555243.2555260.

– (2016). "Concurrency Testing Using Controlled Schedulers: an Empirical Study". In: *ACM Transactions on Parallel Computing* 2.4, 23:1–23:37. DOI: 10.1145/2858651.

Tweag I/O developer (2017). "Email correspondence about a slow test".

Vollmer, Michael et al. (2017). "SC-Haskell: Sequential Consistency in Languages That Minimize Mutable Shared Heap". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP '17. ACM, pp. 283–298. DOI: 10.1145/3018743.3018746.

Walker, Michael (2016). *Déjà Fu: A Concurrency Testing Library for Haskell.* Tech. rep. University of York, Department of Computer Science.

– (2018a). "hunit-dejafu: Deja Fu support for the HUnit test framework". At https://hackage.haskell.org/package/hunit-dejafu, accessed at 2018-01-25.

– (2018b). "tasty-dejafu: Deja Fu support for the Tasty test framework". At https://hackage.haskell.org/package/tasty-dejafu, accessed at 2018-01-25.

Walker, Michael and Colin Runciman (2015). "Déjà Fu: A Concurrency Testing Library for Haskell". In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell.* Haskell 2015. ACM, pp. 141–152. DOI: 10.1145/2804302.2804306.

– (2017). “Cheap Remarks about Concurrent Programs”.
Presented at: *Trends in Functional Programming.*

– (2018). “Cheap Remarks About Concurrent Programs”.
In: *Functional and Logic Programming.*
Ed. by John P. Gallagher and Martin Sulzmann. FLOPS 2018.
Springer International Publishing, pp. 264–279.

Woo, Steven Cameron et al. (1995).
“The SPLASH-2 Programs: Characterization and Methodological Considerations”.
In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*.
ISCA ’95. ACM, pp. 24–36. DOI: 10.1145/223982.223990.

Yang, Edward Z. (2017). “Backpack: Towards Practical Mix-in Linking in Haskell”.
PhD thesis.

Yang, Yu, Xiaofang Chen, and Ganesh Gopalakrishnan (2008).
Inspect: A Runtime Model Checker for Multithreaded C Programs. Tech. rep.
University of Utah.

Yu, Jie et al. (2012).
“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. ACM, pp. 485–502.
DOI: 10.1145/2398857.2384651.

Zhang, Naling, Markus Kusano, and Chao Wang (2015).
“Dynamic Partial Order Reduction for Relaxed Memory Models”.
In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. ACM, pp. 250–259.
DOI: 10.1145/2737924.2737956.