ZuriHac Advanced Track - IOSim

Armando Santos

June 8, 2022 —

About me

Figure: ↑ Me

- Haskell Consultant at **Well-Typed**
- Working in the Networking Team of IOHK
- Formal Methods & Distributed Systems Enthusiast

You can find me at bolt12 and _bolt12 !

Well-Typed

ZuriHac Advanced Track - IOSim

Materials for the Session

TODO...

What is IOSim?

- Drop-in replacement for IO in Haskell
- A set of IO typeclasses for each different capability
- A tool and a framework to analyse simulated execution trace

Figure: Help

Where and How is it used?

Where:

- IOHK Networking team;
- Soon in someone's personal project (hopefully!).

How:

- Run same user code in simulation and production;
- Simulate high and low level components of the ouroboros-network stack:
  - Test for high level properties of such components;
  - Find edge case bugs;
  - Simulate execution for years worth of simulated time.
- Crazy use case in someone's personal project (hopefully!)

Where to find?

→ **https://github.com/input-output-hk/io-sim** ←

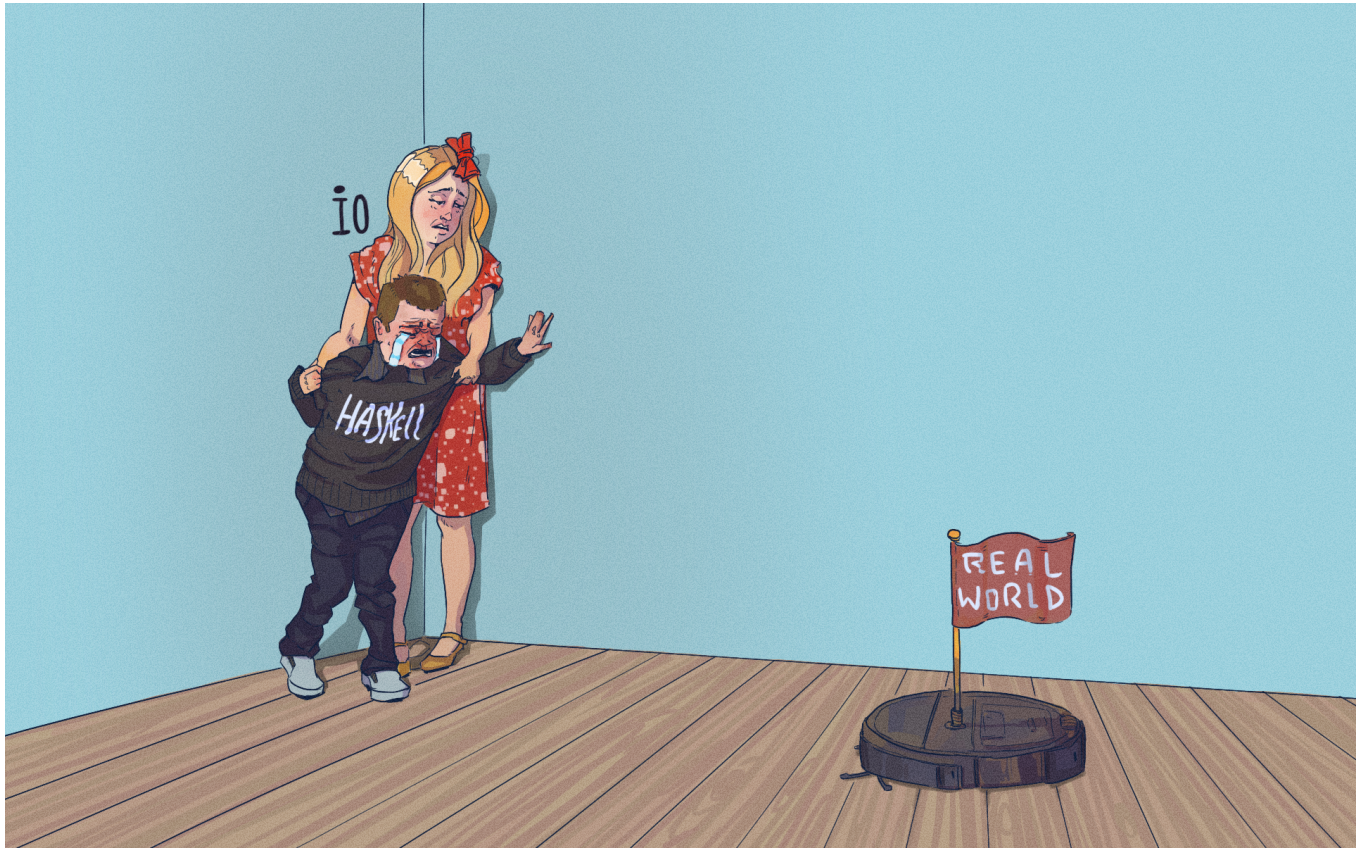


Figure: Real World

About this Workshop

What we will cover

This session is going to try and cover all the basics in order for you to be able to use IOSim in a personal project. That includes:

- Getting a project up and running: **here!**;
- Intro to (contravariant) logging;
- Look into how IOSim works;
- Look into testing;
- Put everything together in a small project!

Contravariant Logging

contra-tracer

There are plenty of nice tutorials about contravariant logging:

- **Kowainik Blog Post**
- **Duncan Coutts MuniHac 2020 Talk**
- etc . . .

We are going to cover the basics.

Why Logging?

- All the reasons you might be familiar with already;
- Contravariant logging just seems a good approach in general;
- With IOSim we are going to need to build a useful execution trace;
- Logging will give us a way to build the events we care about into the trace.

Figure: Reuse

contra-tracer

```haskell
newtype Tracer m a = Tracer (a -> m ())
```

```haskell
traceWith :: Tracer m a -> a -> m ()
traceWith (Trace t) x = t x
```

```haskell
instance Contravariant (Tracer m) where
  contramap f (Tracer t) = Tracer (t . f)
```

contra-tracer examples

```
nullTracer :: Applicative m => Tracer m a
nullTracer = Tracer (\_ -> pure ())

stdoutTracer :: Tracer IO String
stdoutTracer = Tracer putStrLn
```

In IOSim we have:

```
traceM :: Typeable a => a -> IOSim s ()
traceM x = IOSim $ \ k -> Output (toDyn x) (k ())
```

io-sim & io-classes

io-sim

`io-sim` is a library that supports:

- Asynchronous exceptions
- Simulated time
- Timeout API
- Software Transaction Memory (STM)
- Concurrency: both low level `forkIO` as well as async style
- Strict STM
- Etc ...

io-sim

`io-sim` is a library that offers:

- `IOSim` data type;
- Functions to manipulate the trace;
- Pretty printers;
- `IOSimPOR` (Partial Order Reduction)

io-classes

`io-classes` is a library that offers a monad class hierarchy that is meant to be an interface between `io-sim` and `IO`.

It aims to offer a subset of simulated `IO` capabilities without altering their original semantics:

- `MonadAsync`
- `MonadEventlog`
- `MonadFork`
- `MonadST`
- `MonadSTM`
- `MonadSay`
- `MonadTest`
- `MonadThrow`
- `MonadTime`
- `MonadTimer`

Figure: Inception

Example in IO

```haskell
data TraceExample = SettingThreadDelay DiffTime
   | Printing String
  deriving (Show)
```

```haskell
exampleIO :: Tracer IO TraceExample -> IO ()
exampleIO trace = do
  traceWith trace (SettingThreadDelay 1000)
  threadDelay 1000
  traceWith trace (Printing "Hello World")
  putStrLn "Hello World"
```

Example in IOSim

```haskell
data TraceExample = SettingThreadDelay DiffTime
   | Printing String
  deriving (Show)
```

```haskell
exampleIOSim :: (MonadDelay m, MonadSay m)
   => Tracer m TraceExample -> m ()
exampleIOSim trace = do
  traceWith trace (SettingThreadDelay 1000)
  threadDelay 1000
  traceWith trace (Printing "Hello World")
  say "Hello World"
```

Main

```
Time 0 s - ThreadId []    main - EventLog << TraceExample >>
Time 0 s - ThreadId []    main - EventSay "SettingThreadDelay 1000s"
Time 0 s - ThreadId []    main - EventTimerCreated (TimeoutId 0)
                           (TVarId 0)
                           (Time 1000 s)
Time 0 s - ThreadId []    main - EventTxBlocked [Labelled (TVarId 0)
                           (Just "<<timeout-state 0>>")]
                           Nothing
Time 0 s - ThreadId []    main - EventDeschedule Blocked
Time 1000 s - ThreadId [- 1] timer - EventTimerExpired (TimeoutId 0)
Time 1000 s - ThreadId [] main - EventTxWakeup [Labelled (TVarId 0)
                           (Just "<<timeout-state 0>>")]
Time 1000 s - ThreadId [] main - EventTxCommitted [] [] Nothing
Time 1000 s - ThreadId [] main - EventUnblocked []
Time 1000 s - ThreadId [] main - EventDeschedule Yield
Time 1000 s - ThreadId [] main - EventLog << TraceExample >>
Time 1000 s - ThreadId [] main - EventSay "Printing \"Hello World\""
Time 1000 s - ThreadId [] main - EventSay "Hello World"
Time 1000 s - ThreadId [] main - EventThreadFinished
MainReturn (Time 1000 s) () []
```

Testing

How to test simulated code?

IOSim API offers the following signatures:

```
runSim :: forall a . (forall s . IOSim s a)
   -> Either Failure a
```

```
runSimOrThrow :: forall a . (forall s . IOSim s a)
   -> a
```

```
runSimStrictShutdown :: forall a . (forall s . IOSim s a)
   -> Either Failure a
```

```
runSimTrace :: forall a . (forall s . IOSim s a)
   -> SimTrace a
```

```
traceResult :: Bool
   -> SimTrace a
   -> Either Failure a
```

```
traceEvents :: SimTrace a
   -> [(Time, ThreadId, Maybe ThreadLabel, SimEventType)]
```

How to test simulated code?

There's also a family of function that allow you to extract only the events you care about:

```
traceSelectTraceEvents :: (SimEventType -> Maybe b)
   -> SimTrace a
   -> Trace (SimResult a) b
```

```
traceSelectTraceEventsDynamic :: forall a b . Typeable b
   => SimTrace a -> Trace (SimResult a) b
```

```
traceSelectTraceEventsSay :: forall a . SimTrace a
   -> Trace (SimResult a) String
```

There's also the `List` variants if needed.

Things to be aware of

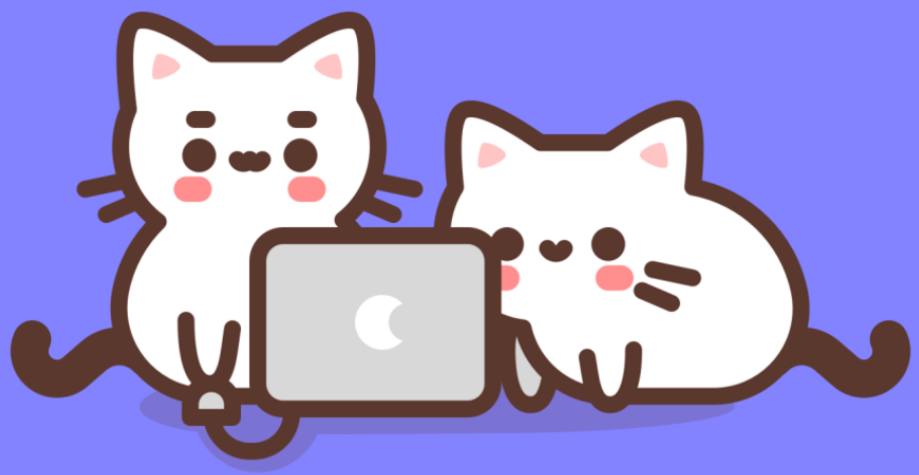
A couple of things to have in mind when using IOSim:

- It is deterministic;
- Time only passes if there are explicit `threadDelays`;
- Trace is as lazy as it can be;
- It is not a model checker;

Figure: Cat pointing

Pause?

Demo



Source code

io-sim

- `CommonTypes.hs`
  Common types shared between `IOSim` and `IOSimPOR`. Like `TVar` and `ThreadId`.
- `Internal.hs`
  `Thread` and `SimState` definitions. Functions to `schedule` and `deschedule` threads are implemented here.
- `InternalTypes.hs`
  Internal types shared between 'IOSim' and 'IOSimPOR'. Like the `ControlStack`, needed to deal with exceptions and masking.

io-sim

- `STM.hs`
  io-sim implementation of TQueue and TBQueue.
- `Types.hs`
  IOSim internal DSL for actions (`SimA` type) such as Timer, Exception, Forking, et.. API. Has also the `StmA` type that is responsible for the STM capabilities. All the **io-classes** instance implementations are defined here.

`IOSim` outer module also exports a lot of utilities to work with `SimTrace` as we have seen.

Small project

Project Prompt

Classic Readers-Writers Problem

Basically have a shared memory area where different writer threads might write to (one at the time), and have any number of reader threads reading from, assuming no writer is writing to the the shared memory area. Useful properties that we ought to test:

- No reader should be able to read while someone is writing
- No 2 writers should be able to write at the same time
- If the last writer writes X then the first reader must read X
- More . . . ?

Some code

This:

```
x + y
  + z
```

. . . or that:

```
\ y -> let x = 2 in x * y
```

And more …

More code

```
x = 2
y = 3

test = x + y
```

With these definitions, `test` evaluates to `5`.