IOSim and Partial Order Reduction

Marcin Szamotulski

24th February 2024

What is IOSim?

`IOSim` is a simulator monad that supports:

- asynchronous exceptions (including masking)
- simulated time
- timeout API
- software transaction memory (STM)
- concurrency: both low-level `forkIO` as well as `async` style
- strict STM
- access to lazy ST
- **schedule discovery**
- event log
- dynamic tracing
- tracing committed changes to `TVar`, `TMVars`, etc.
- labeling of threads, `TVar`'s, etc.

io-classes

io-classes provide class based monad polymorphic api which allows to write code which can be executed both in `IO` and `IOSim`.

```
withAsyncs :: MonadAsync m
           => [m a]
           -> ([Async m a] -> m b)
           -> m b
withAsyncs xs0 action = go [] xs0
  where
    go as []     = action (reverse as)
    go as (x:xs) = withAsync x (\a -> go (a:as) xs)
```

We also developed a few extensions which are packaged as a seprate libraries: strict-stm, strict-mvar, si-timers.

IOSim - trace

```
sim :: (MonadLabelledSTM m,
        MonadTimer m,
        MonadTraceSTM m,
        MonadSay m) => m ()
sim = do
    d <- registerDelay 1_000_000
    labelTVarIO d "delayVar"
    traceTVarIO d (\_ a -> pure (TraceString (show a)))
    atomically (readTVar d >>= check)
    say "Arr, land ho!"
```

IOSim - trace

```haskell
sim :: (MonadLabelledSTM m,
        MonadTimer m,
        MonadTraceSTM m,
        MonadSay m) => m ()
sim = do
    d <- registerDelay 1_000_000
    labelTVarIO d "delayVar"
    traceTVarIO d (\_ a -> pure (TraceString (show a)))
    atomically (readTVar d >>= check)
    say "Arr, land ho!"
```

```
0s - Thread []     main - RegisterDelayCreated TimeoutId 0 TVarId 0 Time 1s
0s - Thread []     main - TxBlocked [Labelled TVarId 0 delayVar]
0s - Thread []     main - Deschedule Blocked BlockedOnSTM
```

IOSim - trace

```haskell
sim :: (MonadLabelledSTM m,
        MonadTimer m,
        MonadTraceSTM m,
        MonadSay m) => m ()
sim = do
    d <- registerDelay 1_000_000
    labelTVarIO d "delayVar"
    traceTVarIO d (\_ a -> pure (TraceString (show a)))
    atomically (readTVar d >>= check)
    say "Arr, land ho!"
```

```
0s - Thread []     main - RegisterDelayCreated TimeoutId 0 TVarId 0 Time 1s
0s - Thread []     main - TxBlocked [Labelled TVarId 0 delayVar]
0s - Thread []     main - Deschedule Blocked BlockedOnSTM


1s - Thread [-1]   register delay timer - Say True
1s - Thread [-1]   register delay timer - RegisterDelayFired TimeoutId 0
```

IOSim - trace

```haskell
sim :: (MonadLabelledSTM m,
        MonadTimer m,
        MonadTraceSTM m,
        MonadSay m) => m ()
sim = do
    d <- registerDelay 1_000_000
    labelTVarIO d "delayVar"
    traceTVarIO d (\_ a -> pure (TraceString (show a)))
    atomically (readTVar d >>= check)
    say "Arr, land ho!"
```

```
0s - Thread []     main - RegisterDelayCreated TimeoutId 0 TVarId 0 Time 1s
0s - Thread []     main - TxBlocked [Labelled TVarId 0 delayVar]
0s - Thread []     main - Deschedule Blocked BlockedOnSTM


1s - Thread [-1]   register delay timer - Say True
1s - Thread [-1]   register delay timer - RegisterDelayFired TimeoutId 0


1s - Thread []     main - TxWakeup [Labelled TVarId 0 delayVar]
1s - Thread []     main - TxCommitted [] []
1s - Thread []     main - Unblocked []
1s - Thread []     main - Deschedule Yield
1s - Thread []     main - Say Arr, land ho!
1s - Thread []     main - ThreadFinished
1s - Thread []     main - MainReturn () []
```

Partial Order Reduction

- segment execution into execution steps, e.g. an `STM` action
- deterministic scheduling policy
- discovery of execution races which depends on execution steps partial order
- techniques to only run executions which can lead to new program states
- instrumentation to follow discovered schedules

Execution Step

```haskell
data Step = Step {
    stepThreadId :: IOSimThreadId,
    stepStep     :: Int,
    stepEffect   :: Effect,
    -- ^ which effects where executed by this steps, e.g.
    -- `TVar` reads / writes, forks, throws or wakeups.
    stepVClock   :: VectorClock
    -- ^ vector clock of the thread at the time when
    -- the step was executed.
  }
  deriving Show
```

`IOSimPOR` thread scheduler will run one thread at a time, and collect `Step` for the period while the thread is being executed.

Execution Step

Life cycle of a `Step`

- when a thread is descheduled:
  - forking a new thread
  - thread termination
  - setting the masking state to interruptible
  - popping masking frame (which resets masking state)
  - thread delays
  - execution of an STM transaction
  - blocking `throwTo`
- throw an exception when there's a corresponding catch frame (i.e. catch handler)

Execution Step

Effect

```
data Effect = Effect {
    effectReads  :: Set TVarId,
    effectWrites :: Set TVarId,
    effectForks  :: Set IOSimThreadId,
    effectThrows :: [IOSimThreadId],
    effectWakeup :: Set IOSimThreadId
  }
```

Execution Step

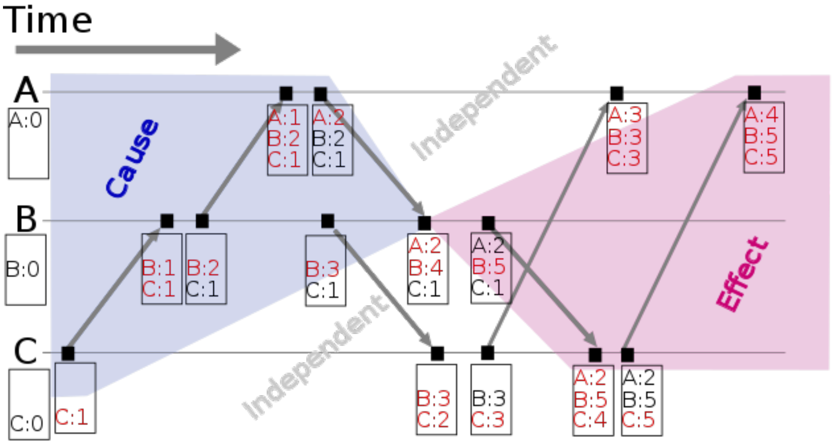
Effect

```haskell
data Effect = Effect {
    effectReads  :: Set TVarId,
    effectWrites :: Set TVarId,
    effectForks  :: Set IOSimThreadId,
    effectThrows :: [IOSimThreadId],
    effectWakeup :: Set IOSimThreadId
  }

racingEffects :: Effect -> Effect -> Bool
racingEffects e e' =
       -- both effects throw to the same threads
       effectThrows e `intersects` effectThrows e'
       -- concurrent reads & writes of the same TVars
    || effectReads  e `intersects`  effectWrites e'
    || effectWrites e `intersects`  effectReads  e'
       -- concurrent writes to the same TVars
    || effectWrites e `intersects`  effectWrites e'
  where
    intersects :: (Foldable f, Eq a) => f a -> f a -> Bool
    intersects a b = not . null $ toList a `List.intersect` toList b
```

Execution Step

Causality



source WikiPedia: Vector Clocks

Extension of Leslie Lamport's logical clocks.

Execution Step

Vector Clocks

```haskell
newtype VectorClock = VectorClock {
    getVectorClock :: Map IOSimThreadId Int
  }
leastUpperBoundVClock :: VectorClock
                      -> VectorClock
                      -> VectorClock
leastUpperBoundVClock (VectorClock m) (VectorClock m') =
    VectorClock (Map.unionWith max m m')
```

For example

```haskell
  ThrowTo e tid' k -> do
    let thread' = thread {
            threadEffect  = effect <> throwToEffect tid'
                                   <> wakeUpEffect,
            threadVClock  =
              vClock `leastUpperBoundVClock` vClockTgt
            ...
          }
        vClockTgt = threadVClock (threads Map.! tid')
```

IOSimPOR Schedule Policy

Run not blocked thread with the smallest `ThreadId`.

```
data IOSimThreadId =
    RacyThreadId [Int]
    -- | A non racy thread.  They have higher priority than
    -- racy threads in `IOSimPOR` scheduler.
  | ThreadId     [Int]

mainThread :: IOSimThreadId
mainThread = ThreadId []


-- second child of `RacyThread [1]`
threadId = RacyThreadId [1,2]
```

As a consequence a thread will be scheduled until it is blocked.

Races

```
data StepInfo = StepInfo {
    -- | Step that we want to reschedule to run after a step in
    -- `stepInfoRaces`.
    stepInfoStep       :: Step,
    -- | Control information when we reach this step.
    stepInfoControl    :: ScheduleControl,
    -- | Threads that are still concurrent with this step.
    stepInfoConcurrent :: Set IOSimThreadId,
    -- | Steps following this one that did not happen after it
    -- (in reverse order).
    stepInfoNonDep     :: [Step],
    -- | Later steps that race with `stepInfoStep`.
    stepInfoRaces      :: [Step] }
```

Races

```haskell
data StepInfo = StepInfo {
    -- | Step that we want to reschedule to run after a step in
    -- `stepInfoRaces`.
    stepInfoStep       :: Step,
    -- | Control information when we reach this step.
    stepInfoControl    :: ScheduleControl,
    -- | Threads that are still concurrent with this step.
    stepInfoConcurrent :: Set IOSimThreadId,
    -- | Steps following this one that did not happen after it
    -- (in reverse order).
    stepInfoNonDep     :: [Step],
    -- | Later steps that race with `stepInfoStep`.
    stepInfoRaces      :: [Step] }
```

New schedules are constructed from `stepInfoRaces` and `stepInfoNonDep`:

```haskell
    [    takeWhile (/=stepStepId racingStep)
                   (stepStepId <$> reverse stepInfoNonDep)
      ++ [stepStepId racingStep]
    | racingStep <- stepInfoRaces ]
```

Races

Recording new StepInfo in active races

```haskell
-- A new step to add to the `activeRaces` list.
newStepInfo :: Maybe StepInfo
newStepInfo | isNotRacyThreadId tid = Nothing
            | Set.null concurrent   = Nothing
            | isBlocking            = Nothing
            | otherwise =
    Just StepInfo { stepInfoStep       = newStep,
                    stepInfoControl    = control,
                    stepInfoConcurrent = concurrent,
                    stepInfoNonDep     = [],
                    stepInfoRaces      = []
                  }
  where
    concurrent :: Set IOSimThreadId
    concurrent = concurrent0 Set.\\ effectWakeup newEffect
    isBlocking :: Bool
    isBlocking = isThreadBlocked thread
              && onlyReadEffect newEffect
```

Races

Updating already recorded active races

With every new step, we need to update existing information recorded in `StepInfo`.

```
let theseStepsRace = step `racingSteps` newStep
    -- `step` happened before `newStep` (`newStep` happened after
    -- `step`)
    happensBefore   = step `happensBeforeStep` newStep
    -- `newStep` happens after any of the racing steps
    afterRacingStep = any (`happensBeforeStep` newStep) stepInfoRaces
```

Races

Updating already recorded active races

With every new step, we need to update existing information recorded in `StepInfo`.

```
let theseStepsRace = step `racingSteps` newStep
    -- `step` happened before `newStep` (`newStep` happened after
    -- `step`)
    happensBefore   = step `happensBeforeStep` newStep
    -- `newStep` happens after any of the racing steps
    afterRacingStep = any (`happensBeforeStep` newStep) stepInfoRaces
```

- update `stepInfoConcurrent`

```
let -- We will only record the first race with each thread.
    -- Reversing the first race makes the next race detectable.
    -- Thus we remove a thread from the concurrent set after the
    -- first race.
    concurrent'
      | happensBefore   = Set.delete tid concurrent
                   Set.\\ effectWakeup newEffect
      | theseStepsRace  = Set.delete tid concurrent
      | afterRacingStep = Set.delete tid concurrent
      | otherwise       = concurrent
```

Races

Updating already recorded active races

With every new step, we need to update existing information recorded in `StepInfo`.

```
let theseStepsRace = step `racingSteps` newStep
    -- `step` happened before `newStep` (`newStep` happened after
    -- `step`)
    happensBefore   = step `happensBeforeStep` newStep
    -- `newStep` happens after any of the racing steps
    afterRacingStep = any (`happensBeforeStep` newStep) stepInfoRaces
```

- update `stepInfoConcurrent`
- update `stepInfoNonDep`

```
let stepInfoNonDep'
      -- `newStep` happened after `step`
      | happensBefore =           stepInfoNonDep
      -- `newStep` did not happen after `step`
      | otherwise     = newStep : stepInfoNonDep
```

Races

Updating already recorded active races

With every new step, we need to update existing information recorded in `StepInfo`.

```
let theseStepsRace = step `racingSteps` newStep
    -- `step` happened before `newStep` (`newStep` happened after
    -- `step`)
    happensBefore   = step `happensBeforeStep` newStep
    -- `newStep` happens after any of the racing steps
    afterRacingStep = any (`happensBeforeStep` newStep) stepInfoRaces
```

- update `stepInfoConcurrent`
- update `stepInfoNonDep`
- update `stepInfoRaces`

```
let -- Here we record discovered races.  We only record new
    -- race if we are following the default schedule, to avoid
    -- finding the same race in different parts of the search
    -- space.
    stepInfoRaces'
      | theseStepsRace && isDefaultSchedule control
                  = newStep : stepInfoRaces
      | otherwise =           stepInfoRaces
```

Example

```haskell
sim :: IOSim s ()
sim = do
  exploreRaces
  v <- newTVarIO False
  forkIO (atomically $ writeTVar v True)
  forkIO (readTVarIO v >>= say . show)
  -- wait for both threads to terminate.
  threadDelay 1_000_000

quickCheck $ exploreSimTrace
      (\a -> a { explorationDebugLevel = 1 })
      sim
      (\_ _ -> True)
```

Example: default schedule

[].0 `create TVar 0`

```
0s - Thread [].0 main - SimStart ControlDefault
0s - Thread [].0 main - TxCommitted [] [TVarId 0] Effect {  }
0s - Thread [].0 main - Unblocked []
0s - Thread [].0 main - Deschedule Yield
0s - Thread [].0 main - Effect VectorClock [Thread [].0]
                               Effect {  }
```

Example: default schedule

[].0 `create TVar 0`

↓

[].1 fork `Thread {1}`

```
0s - Thread [].1 main - ThreadForked Thread {1}
0s - Thread [].1 main - Deschedule Yield
0s - Thread [].1 main - Effect VectorClock [Thread [].1]
                                 Effect { forks = [Thread {1}] }
```

Example: default schedule

[].0 create `TVar 0`

↓

[].1

↓

[].2 fork `Thread {2}`

```
0s - Thread [].2 main - ThreadForked Thread {2}
0s - Thread [].2 main - Deschedule Yield
0s - Thread [].2 main - Effect VectorClock [Thread [].2]
                            Effect { forks = [Thread {2}] }
```

Example: default schedule

```
0s - Thread [].3 main - ThreadDelay TimeoutId 0 Time 1s
0s - Thread [].3 main - Effect VectorClock [Thread [].3]
                                 Effect {  }
```

[].0 `create TVar 0`

↓

[].1

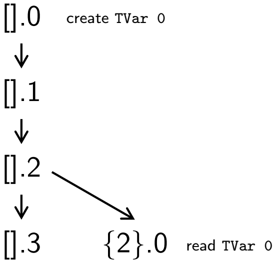
↓

[].2

↓

[].3 `threadDelay` 10^6

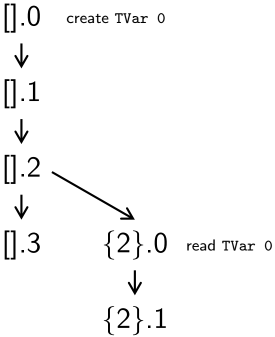
Example: default schedule




```
data StepInfo = StepInfo {
    stepInfoStep       = Step ({2}.0),
    stepInfoControl    = DefaultControl,
    stepInfoConcurrent = Set.fromList
                         [[], {1}, {2}],
    stepInfoNonDep     = [],
    stepInfoRaces      = []
  }
```

```
0s - Thread {2}.0  - TxCommitted [] []
                       Effect { reads = fromList [TVarId 0] }
0s - Thread {2}.0  - Unblocked []
0s - Thread {2}.0  - Deschedule Yield
0s - Thread {2}.0  - Effect VectorClock [Thread {2}.0,
                                         Thread [].2]
                       Effect { reads = fromList [TVarId 0] }
```

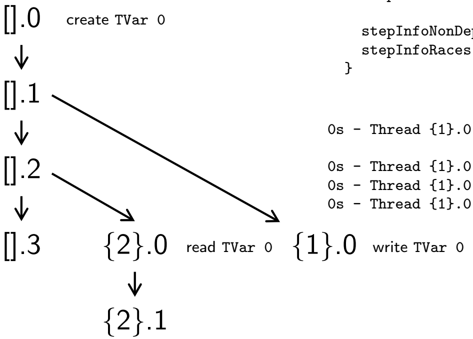
Example: default schedule




```
data StepInfo = StepInfo {
    stepInfoStep       = Step ({2}.0),
    stepInfoControl    = DefaultControl,
    stepInfoConcurrent = Set.fromList
                         [[], {1}],
    stepInfoNonDep     = [],
    stepInfoRaces      = []
  }
```

```
0s - Thread {2}.1  - Say False
0s - Thread {2}.1  - ThreadFinished
0s - Thread {2}.1  - Deschedule Terminated
0s - Thread {2}.1  - Effect VectorClock [Thread {2}.1,
                                         Thread [].2]
                     Effect {  }
```

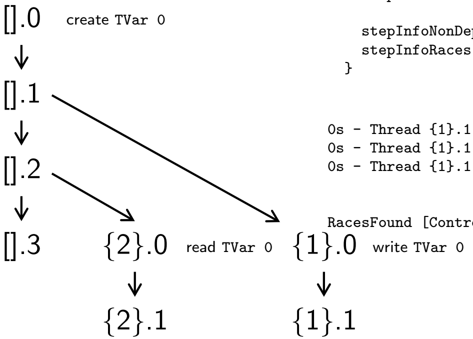
Example: default schedule




```
data StepInfo = StepInfo {
    stepInfoStep       = Step ({2}.0),
    stepInfoControl    = DefaultControl,
    stepInfoConcurrent = Set.fromList
                         [[], {1}],
    stepInfoNonDep     = [{1}.0],
    stepInfoRaces      = [{1}.0]
  }
```

```
0s - Thread {1}.0  - TxCommitted [TVarId 0] []
                      Effect { writes = fromList [TVarId 0] }
0s - Thread {1}.0  - Unblocked []
0s - Thread {1}.0  - Deschedule Yield
0s - Thread {1}.0  - Effect VectorClock [Thread {1}.0,
                                         Thread [].1]
                      Effect { writes = fromList [TVarId 0] }
```

Example: default schedule




```
data StepInfo = StepInfo {
    stepInfoStep       = Step ({2}.0),
    stepInfoControl    = DefaultControl,
    stepInfoConcurrent = Set.fromList
                         [[]],
    stepInfoNonDep     = [{1}.0],
    stepInfoRaces      = [{1}.0]
  }
```

```
0s - Thread {1}.1  - ThreadFinished
0s - Thread {1}.1  - Deschedule Terminated
0s - Thread {1}.1  - Effect VectorClock [Thread {1}.1,
                                         Thread [].1]
                             Effect {  }
RacesFound [ControlAwait [ScheduleMod (RacyThreadId [2],0)
                                      ControlDefault
                                      [(RacyThreadId [1],0)]]]
```

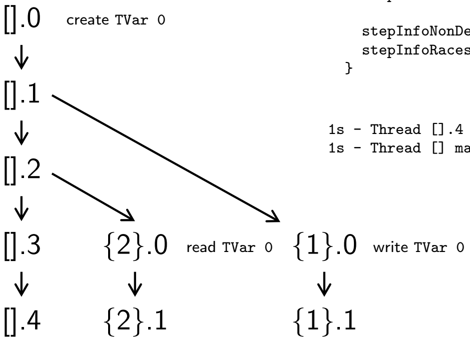
Example: default schedule

$$
\begin{tikzcd}
{[].0}\ \text{create TVar 0} \arrow[d] & & \\
{[].1} \arrow[d] \arrow[ddrr] & & \\
{[].2} \arrow[d] \arrow[dr] & & \\
{[].3} & \{2\}.0\ \text{read TVar 0} \arrow[d] & \{1\}.0\ \text{write TVar 0} \arrow[d] \\
& \{2\}.1 & \{1\}.1
\end{tikzcd}
$$

```
data StepInfo = StepInfo {
    stepInfoStep       = Step ({2}.0),
    stepInfoControl    = DefaultControl,
    stepInfoConcurrent = Set.fromList
                         [[]],
    stepInfoNonDep     = [{1}.0],
    stepInfoRaces      = [{1}.0]
  }
```

```
1s - Thread [].- thread delay timer - ThreadDelayFired
                                          TimeoutId 0
```

Example: default schedule




```
data StepInfo = StepInfo {
    stepInfoStep       = Step ({2}.0),
    stepInfoControl    = DefaultControl,
    stepInfoConcurrent = Set.fromList
                         [[]],
    stepInfoNonDep     = [{1}.0],
    stepInfoRaces      = [{1}.0]
  }
```

```
1s - Thread [].4 main - ThreadFinished
1s - Thread [] main - MainReturn () []
```

Example: discovered schedule

[].0 create TVar 0

```
0s - Thread [].0 main - SimStart ControlAwait
                            [ScheduleMod (RacyThreadId [2],0)
                                         ControlDefault
                                         [(RacyThreadId [1],0)]]
0s - Thread [].0 main - TxCommitted [] [TVarId 0] Effect {  }
0s - Thread [].0 main - Unblocked []
0s - Thread [].0 main - Deschedule Yield
0s - Thread [].0 main - Effect VectorClock [Thread [].0]
                          Effect {  }
```

Example: discovered schedule

[].0 create `TVar 0`

↓

[].1 fork `Thread {1}`

```
0s - Thread [].1 main - ThreadForked Thread {1}
0s - Thread [].1 main - Deschedule Yield
0s - Thread [].1 main - Effect VectorClock [Thread [].1]
                                  Effect { forks = [Thread {1}] }
```

Example: discovered schedule

```
0s - Thread [].2 main - ThreadForked Thread {2}
0s - Thread [].2 main - Deschedule Yield
0s - Thread [].2 main - Effect VectorClock [Thread [].2]
                          Effect { forks = [Thread {2}] }
```

[].0 create `TVar 0`

↓

[].1

↓

[].2 fork `Thread {2}`

Example: discovered schedule

```
0s - Thread [].3 main - ThreadDelay TimeoutId 0 Time 1s
0s - Thread [].3 main - Effect VectorClock [Thread [].3]
                          Effect {  }
```

[].0 `create TVar 0`

↓

[].1

↓

[].2

↓

[].3 `threadDelay` 10^6

Example: discovered schedule

[].0 create TVar 0

↓

[].1

↓

[].2

↓

[].3

```
0s - Thread {2}.0  - FollowControl ControlAwait
                      [ScheduleMod (RacyThreadId [2],0)
                                   ControlDefault
                                   [(RacyThreadId [1],0)]]
0s - Thread {2}.0  - AwaitControl Thread {2}.0 ControlFollow [(R
0s - Thread {2}.0  - Deschedule Sleep
```

Example: discovered schedule

[].0 create TVar 0

[].1

[].2

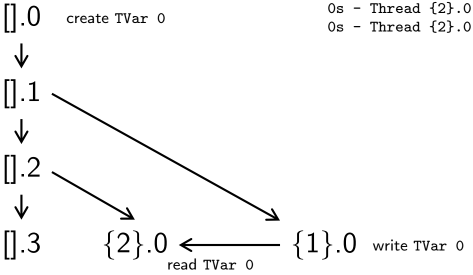
[].3

{1}.0 write TVar 0

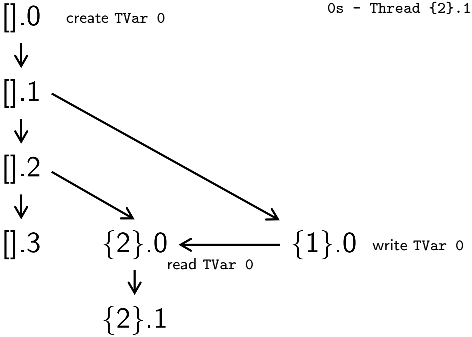
```
0s - Thread {1}.0  - Reschedule ControlFollow
                               [(RacyThreadId [1],0)] []
0s - Thread {1}.0  - PerformAction Thread {1}.0
0s - Thread {1}.0  - TxCommitted [TVarId 0] []
                      Effect { writes = fromList [TVarId 0] }
0s - Thread {1}.0  - Unblocked []
0s - Thread {1}.0  - Deschedule Yield
0s - Thread {1}.0  - Effect
                      VectorClock [Thread {1}.0,
                                   Thread [].1]
                      Effect { writes = fromList [TVarId 0] }
```

Example: discovered schedule

```
0s - Thread {2}.0  - TxCommitted [] []
                      Effect { reads = fromList [TVarId 0] }
0s - Thread {2}.0  - Unblocked []
0s - Thread {2}.0  - Deschedule Yield
0s - Thread {2}.0  - Effect
                     VectorClock [Thread {1}.0,
                                  Thread {2}.0,
                                  Thread [].2]
                     Effect { reads = fromList [TVarId 0] }
```



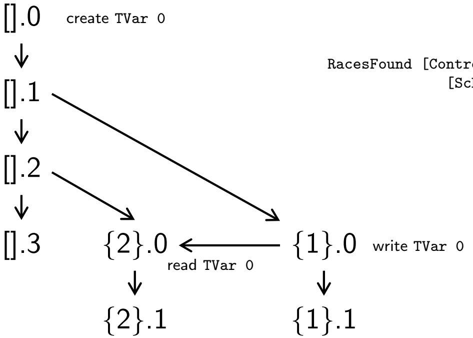
Example: discovered schedule



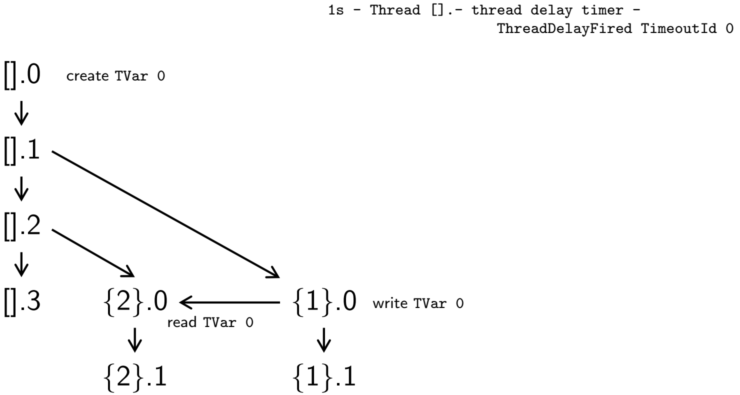

```
0s - Thread {2}.1  - Say True
0s - Thread {2}.1  - ThreadFinished
0s - Thread {2}.1  - Deschedule Terminated
0s - Thread {2}.1  - Effect
                       VectorClock [Thread {1}.0,
                                    Thread {2}.1,
                                    Thread [].2]
                       Effect {  }
```

Example: discovered schedule

```
0s - Thread {1}.1  - ThreadFinished
0s - Thread {1}.1  - Deschedule Terminated
0s - Thread {1}.1  - Effect
                      VectorClock [Thread {1}.1,
                                   Thread [].1]
                      Effect {  }
RacesFound [ControlAwait
              [ScheduleMod (RacyThreadId [2],0)
                           ControlDefault
                           [(RacyThreadId [1],0)]]]
```

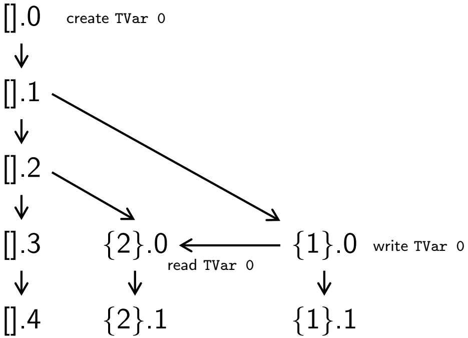


Example: discovered schedule



Example: discovered schedule

```
1s - Thread [].4 main - ThreadFinished
RacesFound []
1s - Thread [] main - MainReturn () []
```



Example 2

```
sim :: IOSim s ()
sim = do
  exploreRaces
  v0 <- newTVarIO False
  v1 <- newTVarIO False
  forkIO (do atomically (writeTVar v0 True) -- Thread {1}.0
             atomically (readTVar  v1)      -- Thread {1}.1
               >>= say . show . ("v1",))
  forkIO (do atomically (writeTVar v1 True) -- Thread {2}.0
             atomically (readTVar  v0)      -- Thread {2}.1
               >>= say . show . ("v0",))
  -- wait for both threads to terminate.
  threadDelay 1_000_000
```

Example 2

Three schedules:

Example 2

Three schedules:

- `ControlDefault`

```
("v0", False)
("v1", True)
```

Example 2

Three schedules:

- `ControlDefault`

  ```
  ("v0", False)
  ("v1", True)
  ```

- `ScheduleMod (RacyThreadId [2],1) ControlDefault
  [(RacyThreadId [1],0)]]`

  ```
  ("v0", True)
  ("v1", True)
  ```

Example 2

Three schedules:

- `ControlDefault`

  ```
  ("v0", False)
  ("v1", True)
  ```

- `ScheduleMod (RacyThreadId [2],1) ControlDefault [(RacyThreadId [1],0)]]`

  ```
  ("v0", True)
  ("v1", True)
  ```

- `ScheduleMod (RacyThreadId [2],0) ControlDefault [(RacyThreadId [1],0),(RacyThreadId [1],1)]`

  ```
  ("v0", True)
  ("v1", False)
  ```

Fair winds and following seas, me mateys!

https://coot.me