async: Asynchronous and Concurrent Programming

The async package provides functionality for performing actions asynchronously, across multiple threads. While it's built on top of the `forkIO` function from base (in `Control.Concurrent`), the async package improves on this in many ways:

- It has graceful and thorough handling of exceptions
- It builds in a way to get results back from a child thread
- There is an `STM` interface for accessing thread results, providing for a convenient way to deal with such things as blocking operatio waiting for results
- Thread cancelation is made easy and reliable
- For some very common use cases, the `race` and `concurrently` functions, as well as the `Concurrently` newtype wrapper, can give you a huge bang for your buck.

Tutorial exercise

To help motivate learning, keep in mind this exercise while reading through the content below, and try to implement a solution. Write a helper function which allows you to pass actions to worker threads, and which properly handles exceptions for all of these actions.

Concepts

There is little cognitive overhead to using this package. The primary datatype it exposes is `Async`. An `Async a` value represents a separate thread which will ultimately generate a value of type `a`. The package follows some pretty standard naming conventions:

- `async*` forks a thread and returns an `Async` value
- `withAsync*` forks a thread and provides the `Async` value to the provided inner action. The forked thread is killed when the inner action exits.
- You can `wait` for the result of an `Async`, `poll` to check if it's complete, or `cancel` it
- By default, `wait`ing will rethrow any exceptions thrown by the forked thread. Use the variants that say `Catch` to catch the exceptions.
- Many wait operations have `STM` variants to make them more easily composable

Concurrently

To warm up, we'll start with examples of using `concurrently`, `race`, and the `Concurrently` newtype. These are simpler (and more efficient) variants of the more general `Async`-based interface. The general rule with this library is: if you can get away with `concurrently`/`race`/`Concurrently`, you should.

Basics

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent
import Control.Concurrent.Async

action1 :: IO Int
action1 = do
    threadDelay 500000 -- just to make it interesting
    return 5

action2 :: IO String
action2 = do
    threadDelay 1000000
    return "action2 result"

main :: IO ()
main = do
    res <- concurrently action1 action2
    print (res :: (Int, String))
```

As you can see, the `concurrently` function waits until both operations complete, and then returns both results in a tuple. In contrast, tl `race` function returns only the first one to complete:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent
import Control.Concurrent.Async

action1 :: IO Int
action1 = do
    threadDelay 500000 -- just to make it interesting
    return 5

action2 :: IO String
action2 = do
    threadDelay 1000000
    return "action2 result"

main :: IO ()
main = do
    res <- race action1 action2
    print (res :: Either Int String)
```

Exercise: what will this program output?

The `Concurrently` newtype wrapper uses the `concurrently` function to implement its `Applicative` instance, and `race` to implement its `Alternative` instance. We can demonstrate that, though the code will be quite a bit more verbose:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Applicative
import Control.Concurrent
import Control.Concurrent.Async

action1 :: IO Int
action1 = do
    threadDelay 500000 -- just to make it interesting
    return 5

action2 :: IO String
action2 = do
    threadDelay 1000000
    return "action2 result"

main :: IO ()
main = do
    res1 <- runConcurrently $ (,)
        <$> Concurrently action1
        <*> Concurrently action2
    print (res1 :: (Int, String))

    res2 <- runConcurrently
          $ (Left <$> Concurrently action1)
        <|> (Right <$> Concurrently action2)
    print (res2 :: Either Int String)
```

While this seems tedious for such an example, the `Concurrently` newtype can be great for larger scale cases, such as when we want tc discard some results.

```
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent.Async
import Data.Foldable (traverse_)

type Score = Int
data Person = Person FilePath Score

people :: [Person]
people =
    [ Person "alice.txt" 50
    , Person "bob.txt" 60
    , Person "charlie.txt" 70
    ]

-- | This function returns a unit value that we don't care about. Using
-- concurrently on two such actions would give us ((), ()).
writePerson :: Person -> IO ()
writePerson (Person fp score) = writeFile fp (show score)

-- | Let's write lots of people to their respective files in parallel, instead
-- of sequentially.
writePeople :: [Person] -> IO ()
writePeople = runConcurrently . traverse_ (Concurrently . writePerson)

-- Note: traverse_ is just mapM_ for Applicative instead instead of Monad.
-- Remember, Concurrently is _not_ a Monad instance.

main :: IO ()
main = writePeople people
```

Exceptions

When either child thread throws an exception, that exception is thrown to the other thread:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent
import Control.Concurrent.Async
import Control.Exception

action1 :: IO Int
action1 = error "action1 errored"

action2 :: IO String
action2 = handle onErr $ do
    threadDelay 500000
    return "action2 completed"
  where
    onErr e = do
        putStrLn $ "action2 was killed by: " ++ displayException e
        throwIO (e :: SomeException)

main :: IO ()
main = do
    res <- concurrently action1 action2
    print res
```

You'll get some interleaving of output most likely since string-based I/O work character-by-character, but you should get the idea from running this.

Exercises:

- Use `Data.Text.IO` instead of string-based I/O to avoid the interleaved output
- Replace `concurrently` with `race`. What result do you get?

Companion infinite threads

There's a neat trick you can accomplish with `race` when you want a companion thread to continue running as long as the main thread is operation:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent
import Control.Concurrent.Async
import Control.Exception

-- | Print successive numbers to stdout. Notice how it returns @a@ instead of
-- @()@. This lets the type system know that, under normal circumstances, this
-- function will never exit.
counter :: IO a
counter =
    let loop i = do
            putStrLn $ "counter: " ++ show i
            threadDelay 1000000
            loop $! i + 1
     in loop 1

-- | This function will continue to run counter with whatever action you've
-- provided, and stop running counter once that action exits. If by some chance
-- counter throws an exception, it will take down your thread as well.
withCounter :: IO a -> IO a
withCounter inner = do
    res <- race counter inner
    case res of
        Left x -> assert False x
        Right x -> return x

-- More succintly
-- withCounter = fmap (either id id) . race counter

main :: IO ()
main = do
    putStrLn "Before withCounter"
    threadDelay 2000000
    withCounter $ do
        threadDelay 2000000
        putStrLn "Inside withCounter"
        threadDelay 2000000
    threadDelay 2000000
    putStrLn "After withCounter"
    threadDelay 2000000
    putStrLn "Exiting!"
```

Exercises:

- Why does the `assert False` never get triggered (aka, why does `race` never return a `Left` value)?
- In the "more succinct" version of the code, how does the `either id id` accomplish the same job as the pattern matching?
- Extra credit: could you replace one of the `id`s in `either id id` with `Data.Void.absurd`?

Advanced While it's nice to be able to run companion threads, it can be restricting to require that your main thread live in `IO`. Perhaps yo
want to have your main thread live in some monad transformer stack on top of `IO` instead. Using the powerful (and complex) monad-control package, we can capture the monadic state to make this work.

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
{-# LANGUAGE FlexibleContexts #-}
import Control.Concurrent
import Control.Concurrent.Async
import Control.Exception
import Control.Monad.Reader
import Control.Monad.Trans.Control

-- | Print successive numbers to stdout. Notice how it returns @a@ instead of
-- @()@. This lets the type system know that, under normal circumstances, this
-- function will never exit.
counter :: IO a
counter =
    let loop i = do
            putStrLn $ "counter: " ++ show i
            threadDelay 1000000
            loop $! i + 1
     in loop 1

-- | This function will continue to run counter with whatever action you've
-- provided, and stop running counter once that action exits. If by some chance
-- counter throws an exception, it will take down your thread as well.
withCounter :: MonadBaseControl IO m => m a -> m a
withCounter inner = control $ \runInIO -> do
    res <- race counter (runInIO inner)
    case res of
        Left x -> assert False x
        Right x -> return x

-- More succintly
-- withCounter = fmap (either id id) . race counter

main :: IO ()
main = do
    putStrLn "Before withCounter"
    threadDelay 2000000
    flip runReaderT "some string" $ withCounter $ do
        liftIO $ threadDelay 2000000
        str <- ask
        liftIO $ putStrLn $ "Inside withCounter, str == " ++ str
        liftIO $ threadDelay 2000000
    threadDelay 2000000
    putStrLn "After withCounter"
    threadDelay 2000000
    putStrLn "Exiting!"
```

Async

Sometimes you need a bit more control than is offered by the `Concurrently` family. In those cases, you'll want to use the `Async` type a its associated functions. The core type for all of this is:

```
data Async a
```

This represents an action running in a different thread which, if successful, will give a result of type a. There are lots of things you can do to interact with that thread:

- See if it's still running
- Wait for it to finish
- Get its result
- Kill it early

Most of this is pretty straightforward, but there are three possibly surprising things worth pointing out right off the bat:

1. I used the phrase "if successful" above. It's possible that the action will instead *fail* with a runtime exception. Therefore, any functior which gets a result also needs to deal with a possible SomeException. As you'll see, a number of the functions here deal with that case by simply rethrowing that exception in the calling thread.
2. In order to allow for more composability (as we'll see later), the ability to query a thread's status can be performed from within an S transaction. Most (if not all) of these STM functions also have IO variants, but that's just for user convenience.
3. Killing a thread early, or canceling it, involves throwing it an async exception. If your action misbehaves with async exceptions, canceling may fail. We'll demonstrate that below, but I recommend checking out the safe-exceptions tutorial for advice on doing thi right.

But before we can get into details, let's just see the launching of some basic Asyncs.

Launching

```
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent (threadDelay)
import Control.Concurrent.Async
import Control.Monad
import Say

talker :: String -> IO ()
talker str = forever $ do
  sayString str
  threadDelay 500000

getResult :: IO Int
getResult = do
  sayString "Doing some big computation..."
  threadDelay 2000000
  sayString "Done!"
  return 42

main :: IO ()
main = do
  async1 <- async $ talker "async"
  withAsync (talker "withAsync") $ \async2 -> do
    async3 <- async getResult

    res <- poll async3
    case res of
      Nothing -> sayString "getResult still running"
      Just (Left e) -> sayString $ "getResult failed: " ++ show e
      Just (Right x) -> sayString $ "getResult finished: " ++ show x

    res <- waitCatch async3
    case res of
      Left e -> sayString $ "getResult failed: " ++ show e
      Right x -> sayString $ "getResult finished: " ++ show x

    res <- wait async3
    sayString $ "getResult finished: " ++ show (res :: Int)

  sayString "withAsync talker should be dead, but not async"
  threadDelay 2000000

  sayString "Now killing async talker"
  cancel async1

  threadDelay 2000000
  sayString "Goodbye!"
```

This demonstrates the two most common ways of launching an `Async`:

- The `async` function will launch a new `Async` without any plans to kill the thread. It's your responsibility to do so if you wish, explicitly, via `cancel`
- `withAsync` will launch a thread and run an action with that thread running. When the supplied action finishes, that thread is

canceled.

The advantage of withAsync is that it is exception safe.

Exercise Implement your own withAsync using bracket, async, and cancel.

This also demonstrated three of the most popular functions for querying an Async's status:

- poll checks if a thread is still running. If it's complete, it gives you a Just with Either the SomeException it failed with, or the result.
- waitCatch will block until the thread exits, and then give you Either the SomeException or the result.
- wait also blocks, but in the case of a SomeException will throw it as a runtime exception.

Exercise Implement wait in terms of waitCatch. Can you efficiently implement waitCatch in terms of poll?

In general, you should prefer withAsync in place of async to avoid having unneeded threads running. If you specifically need a thread t outlive a certain block, then async is the right function to use.

Composing in STM

Let's simplify our example above a bit, and instead of using the IO variants of the functions, use the STM functions. This should look pret familiar:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent (threadDelay)
import Control.Concurrent.Async
import Control.Concurrent.STM
import Control.Monad
import Say

getResult :: IO Int
getResult = do
  sayString "Doing some big computation..."
  threadDelay 2000000
  sayString "Done!"
  return 42

main :: IO ()
main = withAsync getResult $ \a -> do
  res <- atomically $ pollSTM a

  case res of
    Nothing -> sayString "getResult still running"
    Just (Left e) -> sayString $ "getResult failed: " ++ show e
    Just (Right x) -> sayString $ "getResult finished: " ++ show x

  res <- atomically $ waitCatchSTM a
  case res of
    Left e -> sayString $ "getResult failed: " ++ show e
    Right x -> sayString $ "getResult finished: " ++ show x

  res <- atomically $ waitSTM a
  sayString $ "getResult finished: " ++ show (res :: Int)
```

All we've done is append `STM` to the function names and stick an `atomically` at the front. On its own, this isn't too exciting, just informative.

Exercise Implement both `waitCatchSTM` and `waitSTM` in terms of `pollSTM`

Exercise Now implement `pollSTM` in terms of `waitCatchSTM`.

So what's so great about this STM business? For one, it can allow us to do more sophisticated queries, like racing two `Async`s:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Applicative ((<|>))
import Control.Concurrent (threadDelay)
import Control.Concurrent.Async
import Control.Concurrent.STM
import Say

getResult1 :: IO Int
getResult1 = do
  sayString "Doing some big computation..."
  threadDelay 2000000
  sayString "Done!"
  return 42

getResult2 :: IO Int
getResult2 = do
  sayString "Doing some smaller computation..."
  threadDelay 1000000
  sayString "Done!"
  return 41

main :: IO ()
main = do
  res <- withAsync getResult1 $ \a1 ->
         withAsync getResult2 $ \a2 ->
         atomically $ waitSTM a1 <|> waitSTM a2

  sayString $ "getResult finished: " ++ show (res :: Int)
```

Yes, in this case using the `race` function would make more sense, but as problems grow in complexity, this flexibility will be important.

Exercise Identify the behavior of this program if `getResult2` throws an exception. Can you modify the code so that, if either of the threads completes successfully, we get a result?

Breaking async exceptions

Look at how easy it is to break our program completely:

```
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
import Control.Concurrent (threadDelay)
import Control.Concurrent.Async
import Control.Exception
import Control.Monad
import Say

evil :: IO ()
evil = forever $ do
  eres <- try $ threadDelay 1000000
  sayShow (eres :: Either SomeException ())

main :: IO ()
main = withAsync evil $ const $ return ()
```

This code will loop forever, since the `cancel` call's exception is caught by the `try` and execution continues indefinitely. This is *very bad code*, don't do this!

Exercise Switch the import from `Control.Exception` to `UnliftIO.Exception`. What happens? Why?

Linking

Let's say you want to run some kind of a background processing thread. You want to kick it off, leave it running, and essentially forget all about it. However, if that thread goes down for some reason, you need to ensure that your main thread goes down too. Such a situation makes a lot of sense, for example, with a job queue.

To make this work, you can use the `link` function, which ensures that if your `Async` ends with an exception, that exception is rethrown t the main thread.

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
{-# LANGUAGE OverloadedStrings #-}
import Control.Concurrent (threadDelay)
import Control.Concurrent.Async
import Control.Concurrent.STM
import Control.Monad (forever)
import Say (say)
import Data.Text (Text)

data Work = Work Text -- intentionally lazy, you'll see why below

jobQueue :: TChan Work -> IO a
jobQueue chan = forever $ do
  Work t <- atomically $ readTChan chan
  say t

main :: IO ()
main = do
  chan <- newTChanIO
  a <- async $ jobQueue chan
  link a
  forever $ do
    atomically $ do
      writeTChan chan $ Work "Hello"
      writeTChan chan $ Work undefined
      writeTChan chan $ Work "World"
    threadDelay 1000000
```

Question What's the behavior without the `link` call?

Question What's the behavior if `Work` was strict in its `Text`? How about if we forced evaluation with `writeTChan chan $! Work $! undefined`?

Exercise Rewrite this program to use `race` instead of `async` and `link`.

Lifted

All of the functions in `Control.Concurrent.Async` live in either the `STM` or `IO` types. Many of the `IO` functions—like `cancel`—could be lifted to `MonadIO` with a simple `liftIO` call. However, many others, like `async`, cannot (since they have the `IO` type appearing as an input, otherwise known as *negative position*). That would seem to exclude the possibility of using monad transformers.

We can make usage of a transformer like `ReaderT` work by manually wrapping/unwrapping its constructor. Let's adapt the `jobQueue` example from above to use a `ReaderT` and see how that goes:

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
{-# LANGUAGE OverloadedStrings #-}
import Control.Concurrent (threadDelay)
import Control.Concurrent.Async
import Control.Concurrent.STM
import Control.Monad (forever)
import Say (say)
import Data.Text (Text)
import Control.Monad.Reader

data Work = Work Text

jobQueue :: ReaderT (TChan Work) IO a
jobQueue = forever $ do
  chan <- ask
  Work t <- liftIO $ atomically $ readTChan chan
  say t

inner :: ReaderT (TChan Work) IO ()
inner = ReaderT $ \chan -> do
  race_ (runReaderT jobQueue chan) $ flip runReaderT chan $ do
    forever $ do
      chan <- ask
      liftIO $ atomically $ do
        writeTChan chan $ Work "Hello"
        writeTChan chan $ Work undefined
        writeTChan chan $ Work "World"
      liftIO $ threadDelay 1000000

main :: IO ()
main = do
  chan <- newTChanIO
  runReaderT inner chan
```

Workable, but tedious. Fortunately, the `unliftio` package provides a version of the async API which is lifted to many more monads. Let see how a simple change in import lets us write much nicer code.

```haskell
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
{-# LANGUAGE OverloadedStrings #-}
import UnliftIO.Async
import UnliftIO.Concurrent (threadDelay)
import UnliftIO.STM
import Control.Monad (forever)
import Say (say)
import Data.Text (Text)
import Control.Monad.Reader

data Work = Work Text

jobQueue :: ReaderT (TChan Work) IO a
jobQueue = forever $ do
  chan <- ask
  Work t <- liftIO $ atomically $ readTChan chan
  say t

inner :: ReaderT (TChan Work) IO ()
inner = do
  race_ jobQueue $ do
    forever $ do
      chan <- ask
      liftIO $ atomically $ do
        writeTChan chan $ Work "Hello"
        writeTChan chan $ Work undefined
        writeTChan chan $ Work "World"
      liftIO $ threadDelay 1000000

main :: IO ()
main = do
  chan <- newTChanIO
  runReaderT inner chan
```

All of the `runReaderT`/`ReaderT` mess in `inner` completely disappeared.

This will work for any monad stack which is an instance of `MonadUnliftIO`, which allows for transformers like `ReaderT` or `IdentityT` but disallows transformers like `StateT` and `WriterT` as they will result in discarded monadic state. Imagine a function like `concurrently (put 5) (put 6)`. Which state will survive? It's frankly arbitrary, and there are three valid options to consider:

- The first one
- The second one
- Discard both states

Using `UnliftIO.Async` as a drop in replacement for `Control.Concurrent.Async` is straightforward, the only API change to be aware of is that the `Async` data type will take an extra type paramter for the underlying monad. This API is also exported directly from th `RIO` module, so if you're using `rio`, you're all set!

Section exercises

Modify the program below so that it completes in under 10 seconds, without modifying `expensiveComputation`:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO
import Test.Hspec
import Control.Monad (replicateM)

data App = App
  { appCounter :: !(TVar Int)
  }

expensiveComputation :: RIO App Int
expensiveComputation = do
  app <- ask
  -- I'm not actually an expensive computation, but I play one on TV
  delayVar <- registerDelay 1000000
  atomically $ do
    readTVar delayVar >>= checkSTM
    let var = appCounter app
    modifyTVar' var (+ 1)
    readTVar var

main :: IO ()
main = hspec $ it "works" $ do
  app <- App <$> newTVarIO 0
  res <- runRIO app $ replicateM 10 expensiveComputation
  sum res `shouldBe` sum [1..10]
```

QUESTION Why is the `sum` necessary to get the tests to pass?

Next, modify the following similar program so that the test passes in under 10 seconds, by only changing the line specified:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO
import Test.Hspec
import Control.Monad (replicateM)

expensiveComputation :: MonadIO m => Int -> m Int
expensiveComputation input = do
  threadDelay 1000000
  pure input

main :: IO ()
main = hspec $ it "works" $ do
  -- Modify only the following line
  res <- for [1..10] expensiveComputation

  -- Don't change this, it's cheating! :)
  res `shouldBe` [1..10]
```

QUESTION Why is `sum` not necessary in this code?

Next, implement the `sumConcurrently` function below.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO
import Test.Hspec
import Control.Monad (replicateM)

expensiveComputation :: MonadIO m => Int -> m Int
expensiveComputation input = do
  threadDelay 1000000
  pure input

sumConcurrently
  :: (a -> IO Int)
  -> [a]
  -> IO Int
sumConcurrently f list = _

main :: IO ()
main = hspec $ it "works" $ do
  res <- sumConcurrently expensiveComputation [1..10]
  res `shouldBe` sum [1..10]
```

SOLUTION There are a few ways of implementing `sumConcurrently`. However, one way that avoids unnecessary lists by leveraging commutativity of addition is:

```haskell
sumConcurrently
  :: (a -> IO Int)
  -> [a]
  -> IO Int
sumConcurrently f list = do
  totalVar <- newTVarIO 0
  forConcurrently_ list $ \a -> do
    int <- f a
    atomically $ modifyTVar' totalVar (+ int)
  atomically $ readTVar totalVar
```

Another set of exercises: implement the `replicateConcurrently_` function using:

- `mapConcurrently_`
- The `Concurrently` newtype wrapper
- `concurrently_`
- `withAsync`

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import Test.Hspec

-- Use mapConcurrently_ here
rc1 :: Int -> IO () -> IO ()
rc1 = undefined

-- Use the Concurrently newtype here
rc2 :: Int -> IO () -> IO ()
rc2 = undefined

-- Use concurrently_ here
rc3 :: Int -> IO () -> IO ()
rc3 = undefined

-- Use withAsync here
rc4 :: Int -> IO () -> IO ()
rc4 = undefined

main :: IO ()
main = hspec $ do
  let rcs =
        [ ("replicateConcurrently_", replicateConcurrently_)
        , ("using mapConcurrently_", rc1)
        , ("using Concurrently (the newtype)", rc2)
        , ("using concurrently_ (the function)", rc3)
        , ("using withAsync", rc4)
        ]
  for_ rcs $ \(name, rc) -> it name $ do
    resVar <- newTVarIO 0
    rc 10 $ atomically $ modifyTVar' resVar (+ 1)
    atomically (readTVar resVar) `shouldReturn` 10
```

SOLUTION

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import Test.Hspec

-- Use mapConcurrently_ here
rc1 :: Int -> IO () -> IO ()
rc1 count action = mapConcurrently_ (const action) [1..count]

-- Use the Concurrently newtype here
rc2 :: Int -> IO () -> IO ()
rc2 count action = runConcurrently $ sequenceA_ $ replicate count (Concurrently action)

-- Use concurrently_ here
rc3 :: Int -> IO () -> IO ()
rc3 count0 action =
    let loop 1 = action
        loop count = concurrently_ action $ loop $ count - 1
     in loop count0

-- Use withAsync here
rc4 :: Int -> IO () -> IO ()
rc4 count0 action =
    let loop 1 = action
        loop count = withAsync action $ \thread -> do
          loop $ count - 1
          wait thread
     in loop count0

main :: IO ()
main = hspec $ do
  let rcs =
        [ ("replicateConcurrently_", replicateConcurrently_)
        , ("using mapConcurrently_", rc1)
        , ("using Concurrently (the newtype)", rc2)
        , ("using concurrently_ (the function)", rc3)
        , ("using withAsync", rc4)
        ]
  for_ rcs $ \(name, rc) -> it name $ do
    resVar <- newTVarIO 0
    rc 10 $ atomically $ modifyTVar' resVar (+ 1)
    atomically (readTVar resVar) `shouldReturn` 10
```

Final exercise: implement `mapConcurrently` using `mapConcurrently_`.

Contact Us

Corporate Office
10130 Perimeter
Parkway

Services

Custom Software
Development
DevSecOps

Products

Kube360®
Zehut
Konsole360

Resources

Blog Posts
Video Library
Case Studies

Our Company

Our Journey
Our Mission
Our Leadership

Suite 200
Charlotte, NC 28216

+1 858-617-0430

sales@fpcomplete.com

Blockchain
Rust
Haskell
Training
All Services

Idiom
Kafka Library

White Papers

Our Engineers
Our Clients
Jobs