-
- Parallel and Concurrent Programming in Haskell

- Comments Off
- Chapters

Table of Contents

Chapter 9. Cancellation and Timeouts

In an interactive application, it is often important for one thread to *interrupt* the execution of another thread after the occurrence of some particular condition. Some examples of this kind of behavior include the following:

- When the user clicks the "stop" button in a web browser, the browser may need to interrupt several activities, such as a thread downloading the page, a thread rendering the page, and a thread running scripts.
- A server application typically wants to give a client a set amount of time to issue a request before closing its connection, so as to avoid letting dormant connections use up resources.
- An application that has a thread running a user interface and a separate thread performing some compute-intensive task (say, generating a visualization of some data) needs to interrupt the computation when the user changes the parameters via the user interface.

The crucial design decision in supporting cancellation is whether the intended victim should have to poll for the cancellation condition or whether the thread is immediately cancelled in some way. This is a tradeoff:

1. If the thread has to poll, then there is a danger that the programmer may forget to poll regularly enough, and the thread will become unresponsive, perhaps permanently so. Unresponsive threads lead to hangs and deadlocks, which are particularly unpleasant from a user’s perspective.
2. If cancellation happens asynchronously, critical sections that modify state need to be protected from cancellation. Otherwise, cancellation may occur mid-update, leaving some data in an inconsistent state.

In fact, the choice is really between doing only (1) or doing both (1) and (2), because if (2) is the default, protecting a critical section amounts to switching to polling behavior for the duration of the critical section.

In most imperative languages, it is unthinkable for (2) to be the default, because so much code modifies state. Haskell has a distinct advantage in this area because most code is purely functional, so it can be safely aborted or suspended and later resumed without affecting correctness. Moreover, our hand is forced: by definition, purely functional code cannot poll for the cancellation condition, so it must be cancellable by default.

Therefore, fully asynchronous cancellation is the only sensible default in Haskell, and the design problem reduces to deciding how cancellation is handled by code in the `IO` monad.

Asynchronous Exceptions

Exceptions are already a fact of life in the `IO` monad, and the usual idioms for writing `IO` monad code include using functions like `bracket` and `finally` to acquire and release resources in a reliable way (see "Exceptions in Haskell"). We would like `bracket` to work even if a thread is cancelled, so cancellation should behave like an exception. However, there's a fundamental difference between the kind of exception thrown by `openFile` when the file does not exist, for example, and an exception that may arise *at any time* because the user pressed the "stop" button. We call the latter kind an *asynchronous* exception because it is asynchronous from the point of view of the "victim"; they didn't ask for it. Conversely, exceptions thrown using the normal `throw` and `throwIO` are called *synchronous* exceptions.

To initiate an asynchronous exception, Haskell provides the `throwTo` primitive, which throws an exception from one thread to another:

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

As with synchronous exceptions, the type of the exception must be an instance of the `Exception` class. The `ThreadId` is a value returned by a previous call to `forkIO`, and may refer to a thread in any state: running, blocked, or finished (in the latter case, `throwTo` is a no-op).

To illustrate the use of `throwTo`, we now elaborate on the example from "Error Handling with Async", in which we downloaded several web pages concurrently, to allow the user to hit `'q'` at any time to stop the downloads.

First, we will extend our `Async` mini-API to allow cancellation. We add one operation:

```
cancel :: Async a -> IO ()
```

This cancels an existing `Async`. If the operation has already completed, then `cancel` has no effect.

To implement `cancel`, we need the `ThreadId` of the thread running the `Async`, so we must store that in the `Async` type along with the `MVar` that holds the result. Hence the `Async` type now looks like:

```
data Async a = Async ThreadId (MVar (Either SomeException a))
```

Given this, the implementation of `cancel` just throws an exception to the thread:

```
cancel :: Async a -> IO ()
cancel (Async t var) = throwTo t ThreadKilled
```

The `ThreadKilled` exception is provided by the `Control.Exception` library and is typically used for cancelling threads in this way.)

For the example, we will need `waitCatch`, which has the same implementation it had in "Error Handling with Async". What happens if we call `waitCatch` on an `Async` that has been cancelled? In that case, `cancel` throws the `ThreadKilled` exception to the thread, so `waitCatch` will return `Left ThreadKilled`.

The remaining piece of the implementation is the `async` operation, which must now store the `ThreadId` returned by `forkIO` in the `Async` constructor:

```
async :: IO a -> IO (Async a)
async action = do
   m <- newEmptyMVar
   t <- forkIO (do r <- try action; putMVar m r)
   return (Async t m)
```

Now we can change the `main` function of the example to support cancelling the downloads:

geturlscancel.hs

```
main = do
  as <- mapM (async . timeDownload) sites                          -- (1)

  forkIO $ do                                                      -- (2)
     hSetBuffering stdin NoBuffering
     forever $ do
        c <- getChar
        when (c == 'q') $ mapM_ cancel as

  rs <- mapM waitCatch as                                          -- (3)
  printf "%d/%d succeeded\n" (length (rights rs)) (length rs) -- (4)
```

(1) Starts the downloads as before.

(2) Forks a new thread that repeatedly reads characters from the standard input and if a `q` is found, calls `cancel` on all the `Async`s.

(3) Waits for all the results (complete or cancelled).

(4) Emits a summary with a count of how many of the operations completed successfully. If we run the sample and hit `q` fast enough, we see something like this:

```
downloaded: http://www.google.com (14538 bytes, 0.17s)
downloaded: http://www.bing.com (24740 bytes, 0.22s)
q2/5 finished
```

Note that this works even though the program is sitting atop a large and complicated HTTP library that provides no direct support for either cancellation or asynchronous I/O. Haskell's support for cancellation is modular in this respect; most library code needs to do nothing to support it, although there are some simple and unintrusive rules that need to be followed when dealing with state, as we shall see in the next section.

Masking Asynchronous Exceptions

As we mentioned earlier, the danger with fully asynchronous exceptions is that one might fire while we are in the middle of updating some shared state, leaving the data in an inconsistent state, and with a high probability of leading to mayhem later. Hence, we certainly need a way to control the delivery of asynchronous exceptions during critical sections. But we must tread carefully: a natural idea is to provide operations to turn off asynchronous exception delivery and turn it on again, but this is not what we really need.

Consider the following problem: a thread wishes to call `takeMVar`, perform an operation depending on the value of the `MVar`, and finally put the result of the operation in the `MVar`. The code must be responsive to asynchronous exceptions, but it should be safe. If an asynchronous exception arrives after the `takeMVar` but before the final `putMVar`, the `MVar` should not be left empty. Instead, the original value should be restored.

If we code this problem using the facilities we've seen so far, we might end up with something like the following function `problem`, which takes two arguments—`m`, an `MVar` to modify, and `f`, a function that takes the current value of the `MVar`—and computes a new value in the `IO` monad.

```
problem :: MVar a -> (a -> IO a) -> IO ()
problem m f = do
  a <- takeMVar m                                 -- ❶
  r <- f a `catch` \e -> do putMVar m a; throw e  -- ❷
  putMVar m r                                     -- ❸
```

There are at least two points where, if an asynchronous exception strikes, the invariant will be violated. If an exception strikes between ❶ and ❷ or between ❷ and ❸, the `MVar` will be left empty. In fact, there is no way to shuffle around the exception handlers to ensure the `MVar` is always left full. To fix this problem, Haskell provides the `mask` combinator:[34]

```
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

The `mask` operation defers the delivery of asynchronous exceptions for the duration of its argument. The type might look a bit confusing, but bear with me. First, I'll show an example of `mask` in use and then explain how it works:[35]

```
problem :: MVar a -> (a -> IO a) -> IO ()
problem m f = mask $ \restore -> do
```

```
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

`mask` is applied to a *function*, which takes as its argument a function `restore`. The `restore` function can be used to restore the delivery of asynchronous exceptions to its present state during execution of the argument to `mask`. If we imagine shading the entire argument to `mask` except for the expression `(f a)`, asynchronous exceptions cannot be raised in the shaded portions.

This solves the problem that we had previously because now an exception can be raised only while `(f a)` is working, and we have an exception handler to catch any exceptions in that case. But a new problem has been introduced: `takeMVar` might block for a long time, but it is inside the `mask` so the thread will be unresponsive during that time. Furthermore, there's no good reason to mask exceptions during `takeMVar`; it would be safe for exceptions to be raised right up until the point where `takeMVar` returns. Hence, this is exactly the behavior that Haskell defines for `takeMVar`: a small number of operations, including `takeMVar`, are designated as *interruptible*. Interruptible operations may receive asynchronous exceptions even inside `mask`.

What justifies this choice? Think of `mask` as "switching to polling mode" for asynchronous exceptions. Inside a `mask`, asynchronous exceptions are no longer asynchronous, but they can still be raised by certain operations. In other words, asynchronous exceptions become *synchronous* inside `mask`.

All operations that may block indefinitely are designated as interruptible.[36] This turns out to be the ideal behavior in many situations, as in the previous `problem` example.

The observant reader may spot a new flaw. The `putMVar` function can also block indefinitely, so the definition of interruptible includes `putMVar`, and therefore the `problem` function above is still unsafe because an asynchronous exception could be raised by either `putMVar`.

However, thanks to a subtlety in the precise definition of interruptibility, we are still safe. An interruptible operation may receive an asynchronous exception only *if it actually blocks*. In the case of `problem` above, we know the `MVar` is definitely empty when we call `putMVar`, so `putMVar` cannot block, which means that it is not interruptible.

How do we know that the `MVar` is definitely empty? Strictly speaking, we don't, because another thread might call `putMVar` on the same `MVar` after the `takeMVar` call in `problem`. The guarantee therefore relies on the `MVar` being operated in a consistent way, where every operation consists of `takeMVar` followed by `putMVar`. This is a common requirement for many `MVar` operations—a particular use of `MVar` comes with a protocol that operations must follow or risk a deadlock.

When you really need to call an interruptible function but can't afford the possibility that an asynchronous exception might be raised, there is a last resort:

```
uninterruptibleMask :: ((IO a -> IO a) -> IO b) -> IO b
```

This works just like `mask`, except that interruptible operations may not receive asynchronous exceptions. Be very careful with `uninterruptibleMask`; accidental misuse may leave your application unresponsive. Every instance of `uninterruptibleMask` should be treated with the utmost suspicion.

For debugging, it is sometimes handy to be able to find out whether the current thread is in the `mask` state or not. The `Control.Exception` library provides a useful function for this purpose:

```
getMaskingState :: IO MaskingState

data MaskingState
  = Unmasked
  | MaskedInterruptible
  | MaskedUninterruptible
```

The `getMaskingState` function returns one of the following constructors:

`Unmasked`
: The current thread is not inside `mask` or `uninterruptibleMask`.

`MaskedInterruptible`
: The current thread is inside `mask`.

`MaskedUninterruptible`
: The current thread is inside `uninterruptibleMask`.

We can provide higher-level combinators to insulate programmers from the need to use `mask` directly. For example, the earlier `problem` function has general applicability when working with `MVar`s and is provided under the name `modifyMVar_` in the `Control.Concurrent.MVar` library:

```
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
```

There is also a variant that allows the operation to return a separate result in addition to the new contents of the `MVar`:

```
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
```

Here's a simple example of `modifyMVar`, used to implement the classic "compare-and-swap" operation:

```
casMVar :: Eq a => MVar a -> a -> a -> IO Bool
casMVar m old new =
  modifyMVar m $ \cur ->
    if cur == old
       then return (new,True)
       else return (cur,False)
```

The `casMVar` function takes an `MVar`, an `old` value, and a `new` value. If the current contents of the `MVar` are equal to `old`, then it is replaced by `new` and `cas` returns `True`; otherwise it is left unmodified and `cas` returns `False`.

Working on multiple `MVar`s is possible by nesting calls to `modifyMVar`. For example, here is a function that modifies the contents of two `MVar`s safely:

modifytwo.hs

```
modifyTwo :: MVar a -> MVar b -> (a -> b -> IO (a,b)) -> IO ()
modifyTwo ma mb f =
  modifyMVar_ mb $ \b ->
    modifyMVar ma $ \a -> f a b
```

If this blocks in the inner `modifyMVar` and an exception is raised, then the outer `modifyMVar_` will restore the contents of the `MVar` it took.

When taking two or more `MVar`s, always take them in the same order. Otherwise, your program is likely to deadlock. We'll discuss this problem in more detail in Chapter 10.

The bracket Operation

We saw the `bracket` function earlier; in fact, `bracket` is defined with `mask` to make it safe in the presence of asynchronous exceptions:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after thing =
  mask $ \restore -> do
    a <- before
    r <- restore (thing a) `onException` after a
    _ <- after a
    return r
```

The `IO` actions passed in as `before` and `after` are performed inside `mask`. The `bracket` function guarantees that if `before` returns, `after` will be executed in the future. It is normal for `before` to contain a blocking operation; if an exception is raised while `before` is blocked, then no harm is done. But `before` should perform only *one* blocking operation. An exception raised by a second blocking operation would not result in `after` being executed. If you need to perform two blocking operations, the right way is to nest calls to `bracket`, as we did with `modifyMVar`.

Something else to watch out for here is using blocking operations in `after`. If you need to do this, then be aware that your blocking operation is interruptible and might receive an asynchronous exception.

Asynchronous Exception Safety for Channels

In most `MVar` code, we can use operations like `modifyMVar_` instead of `takeMVar` and `putMVar` to make our code safe in the presence of asynchronous exceptions. For example, consider the buffered channels that we defined in "MVar as a Building Block: Unbounded Channels". As defined, the operations are not safe in the presence of asynchronous exceptions. For example, `readChan` was defined like this:

```
readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  stream <- takeMVar readVar
  Item val new <- readMVar stream
  putMVar readVar new
  return val
```

If an asynchronous exception occurs after the first `takeMVar`, then the `readVar` will be left empty and subsequent readers of the `Chan` will deadlock. To make it safe, we could use `modifyMVar`:

chan3.hs

```
readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  modifyMVar readVar $ \stream -> do
    Item val tail <- readMVar stream
    return (tail, val)
```

However, this isn't enough on its own. Remember that `readMVar` is defined like this:

```
readMVar :: MVar a -> IO a
readMVar m = do
  a <- takeMVar m
  putMVar m a
  return a
```

So it is possible that an exception arrives between the `takeMVar` and the `putMVar` in `readMVar`, which would leave the `MVar` empty. Hence we also need to use a safe `readMVar` here. There are a few approaches that work. One would be to use `modifyMVar` again to restore the original value. Another approach is to use a variant of `modifyMVar`:

```
withMVar :: MVar a -> (a -> IO b) -> IO b
```

This is like `modifyMVar` but does not change the contents of the `MVar`, and so would be more direct for the purposes of `readMVar`.

The simplest approach, and the one used by the `Control.Concurrent.MVar` library itself, is just to protect `readMVar` with a `mask`:

```
readMVar :: MVar a -> IO a
readMVar m =
  mask_ $ do
    a <- takeMVar m
    putMVar m a
```

```
    return a
```

Here `mask_` is like `mask`, but it doesn't pass a `restore` function. We can get away with this simple definition because unlike `modifyMVar`, there is no operation to perform between the `takeMVar` and `putMVar`, and so no exception handler is required.

With `writeChan`, we have to be a little careful. Here is the original definition:

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newHole <- newEmptyMVar
  oldHole <- takeMVar writeVar
  putMVar oldHole (Item val newHole)
  putMVar writeVar newHole
```

To make the code exception-safe, our first thought might be to try this:

```
wrongWriteChan :: Chan a -> a -> IO ()
wrongWriteChan (Chan _ writeVar) val = do
  newHole <- newEmptyMVar
  modifyMVar_ writeVar $ \oldHole -> do
    putMVar oldHole (Item val newHole)  -- 1
    return newHole                      -- 2
```

But that doesn't work because an asynchronous exception could strike between 1 and 2. This would leave the `old_hole` full and `writeVar` pointing to it, which violates the invariants of the data structure. Hence we need to prevent that possibility too, and the simplest way is just to `mask_` the whole sequence:

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newHole <- newEmptyMVar
  mask_ $ do
    oldHole <- takeMVar writeVar
    putMVar oldHole (Item val newHole)
    putMVar writeVar newHole
```

Note that the two `putMVar`s are both guaranteed not to block, so they are not interruptible.

Timeouts

A useful illustration of programming with asynchronous exceptions is to write a function that can impose a time limit on a given action. We want to provide the timeout wrapper as a combinator of the following type:

```
timeout :: Int -> IO a -> IO (Maybe a)
```

Where `timeout` *`t`* *`m`* has the following behavior:

1. `timeout t m` behaves exactly like `fmap Just m`, if `m` returns a result or raises an exception (including an asynchronous exception) within `t` microseconds.
2. Otherwise, `m` is sent an asynchronous exception of the form `Timeout u`. `Timeout` is a new data type that we define, and `u` is a unique value of type `Unique`, distinguishing this particular instance of `timeout` from any other. The call to `timeout` then returns `Nothing`.

The implementation is not expected to implement real-time semantics, so in practice the timeout will only approximate the requested `t` microseconds. Note that (1) requires that `m` is executed in the context of the current thread because `m` could call `myThreadId`, for example. Also, another thread throwing an exception to the current thread with `throwTo` will expect to interrupt `m`. It should be possible to *nest* timeouts, with the expected behavior.

The code for `timeout`, shown below, was taken from the library `System.Timeout` (with some cosmetic changes for presentation here). The implementation is tricky to get right. The basic idea is to fork a new thread that will wait for `t` microseconds and then call `throwTo` to throw the `Timeout` exception back to the original thread; that much seems straightforward enough. If the operation completes within the time limit, then we must ensure that this thread never throws its `Timeout` exception, so `timeout` must kill the thread before returning.

timeout.hs

```
timeout t m
    | t <  0    = fmap Just m                           -- (1)
    | t == 0    = return Nothing                        -- (2)
    | otherwise = do
        pid <- myThreadId                               -- (3)
        u <- newUnique                                  -- (4)
        let ex = Timeout u                              -- (5)
        handleJust                                      -- (6)
           (\e -> if e == ex then Just () else Nothing) -- (7)
           (\_ -> return Nothing)                       -- (8)
           (bracket (forkIO $ do threadDelay t          -- (9)
                                 throwTo pid ex)
                    (\tid -> throwTo tid ThreadKilled)  -- (10)
                    (\_ -> fmap Just m))                -- (11)
```

Here is how the implementation works, line by line:

(1) (2) Handle the easy cases, where the timeout is negative or zero.

3. Find the `ThreadId` of the current thread.

4. 5. Make a new `Timeout` exception by generating a unique value with `newUnique`.

6. `handleJust` is an exception handler, with the following type:

```
handleJust :: Exception e
           => (e -> Maybe b) -> (b -> IO a) -> IO a
           -> IO a
```

7. The first argument to `handleJust` selects which exceptions to catch. We only want to catch a `Timeout` exception containing the unique value that we created earlier.

8. The second argument to `handleJust` is the exception handler, which in this case returns `Nothing` because timeout occurred.

9. The computation to run inside `handleJust`. Here, we fork the child thread, using `bracket` to ensure that the child thread is always killed before the `timeout` function returns. In the child thread, we wait for `t` microseconds with `threadDelay` and then throw the `Timeout` exception to the parent thread with `throwTo`.

10. Always kill the child thread before returning.

11. The body of `bracket`: run the computation `m` passed in as the second argument to `timeout` and wrap the result in `Just`.

I encourage you to verify that the implementation works by thinking through the two cases: either `m` completes and returns a value, or the child thread throws its exception while `m` is still working.

There is one other tricky case to consider: what happens if *both* the child thread and the parent thread try to call `throwTo` at the same time? Who wins?

The answer depends on the semantics of `throwTo`. In order for this implementation of `timeout` to work properly, the call to `bracket` must not be able to return while the `Timeout` exception can still be thrown; otherwise, the exception can leak. Hence, the call to `throwTo` that kills the child thread must be synchronous. Once this call returns, the child thread cannot throw its exception anymore. Indeed, this guarantee is provided by the semantics of `throwTo`. A call to `throwTo` returns only after the exception has been raised in the target thread. Hence `throwTo` may block if the child thread is currently masking asynchronous exceptions with `mask`, and because `throwTo` may block, it is therefore *interruptible* and may itself receive asynchronous exceptions.

Returning to our "who wins" question above, the answer is "exactly one of them," and that is precisely what we require to ensure the correct behavior of `timeout`.

Catching Asynchronous Exceptions

Once thrown, an asynchronous exception propagates like a normal exception and can be caught by `catch` and the other exception-handling functions from `Control.Exception`. Suppose we catch an asynchronous exception and want to perform some operation as a result, but before we can do that, *another* asynchronous exception is received by the current thread, interrupting the first exception handler. This is undesirable: if asynchronous exceptions can interrupt exception handlers, it is hard to guarantee anything about cleanup actions performed in the event of an exception, for example.

We could fix the problem by wrapping all our calls to `catch` with a `mask` and `restore` pair, like so:

```
  mask $ \restore ->
    restore action `catch` handler
```

And indeed some of our calls to `catch` already look like this. But since we almost always want asynchronous exceptions masked inside an exception handler, Haskell does it automatically for you, without having to use an explicit `mask`. After you return from the exception handler, exceptions are unmasked again.

There is one important pitfall to be aware of here: it is easy to accidentally remain inside the implicit `mask` by tail-calling out of an exception handler. Here's an example program to illustrate the problem: the program takes a list of filenames on the command line and counts the number of lines in each file, ignoring files that do not exist.

catch-mask.hs

```
main = do
  fs <- getArgs
  let
     loop !n [] = return n
     loop !n (f:fs)
        = handle (\e -> if isDoesNotExistError e
                           then loop n fs
```

```
                              else throwIO e) $
            do
               getMaskingState >>= print
               h <- openFile f ReadMode
               s <- hGetContents h
               loop (n + length (lines s)) fs

  n <- loop 0 fs
  print n
```

The `loop` function recursively walks down the list of filenames, attempting to open and read each one, and keeping track of the total lines so far in the first argument `n`. For each filename, first we call `handle` to set up an exception handler. If the exception handler catches an exception that satisfies `isDoesNotExistError` (from `System.IO.Error`), indicating that the file we tried to open did not exist, the exception handler recursively calls `loop` to look at the rest of the files.

This program works, but it has a problem that is revealed by the `getMaskingState` call. Suppose we run the program with a couple of filenames that don't exist:

```
$ ./catch-mask xxx yyy
Unmasked
MaskedInterruptible
0
```

The first time around the loop, we are in the `Unmasked` state, as expected, but the second iteration of `loop` reports that we are now `MaskedInterruptible`! This is clearly suboptimal, because we didn't intend to mask asynchronous exceptions for the second loop iteration.

The problem arose because we made a recursive call to `loop` from the exception handler; thus the recursive call is made inside the implicit `mask` of `handle`.

A better way to code this example is to use `try` instead:

catch-mask2.hs

```
main = do
  fs <- getArgs
  let
     loop !n [] = return n
     loop !n (f:fs) = do
        getMaskingState >>= print
        r <- Control.Exception.try (openFile f ReadMode)
        case r of
          Left e | isDoesNotExistError e -> loop n fs
                 | otherwise             -> throwIO e
          Right h -> do
            s <- hGetContents h
            loop (n + length (lines s)) fs

  n <- loop 0 fs
  print n
```

Now there is no exception handler as such (it is hidden inside `try`), so the recursive call to `loop` is not made within a `mask`. Moreover, we have narrowed the scope of the exception handling to just the `openFile` call, which is neater than before.

However, beware! If you need to handle *asynchronous* exceptions, it's usually important for the exception handler to be inside a `mask` so that you don't get interrupted by another asynchronous exception before you've finished dealing with the first one. For that reason, `catch` or `handle` might be more appropriate, because you can take advantage of the built-in `mask`. Just be careful to return from the exception handler rather than tail-calling out of it, to avoid the problem described above.

mask and forkIO

Let's return to our `Async` API for a moment, and in particular the `async` function:

```
async :: IO a -> IO (Async a)
async action = do
   m <- newEmptyMVar
   t <- forkIO (do r <- try action; putMVar m r)
   return (Async t m)
```

In fact, there's a bug here. If this `Async` is cancelled, and the exception strikes just after the `try` but before the `putMVar`, then the thread will die without putting anything into the `MVar` and the application will deadlock when it tries to `wait` for the result of this `Async`.

We could close this hole with a `mask`, but there's another one: the exception might also arrive just *before* the `try`, with the same consequences. So how do we mask asynchronous exceptions in that small window between the thread being created and the call to `try`? Putting a call to `mask` inside the `forkIO` isn't enough. There is still a possibility that the exception might be thrown even before `mask` is called.

For this reason, `forkIO` is specified to create a thread that *inherits* the masking state of the parent thread. This means that we can create a thread that is born in the masked state by wrapping the call to `forkIO` in a `mask`, for example:

```
async :: IO a -> IO (Async a)
async action = do
   m <- newEmptyMVar
   t <- mask $ \restore ->
          forkIO (do r <- try (restore action); putMVar m r)
   return (Async t m)
```

This pattern of performing some action when a thread has completed is fairly common, so we can embody it as a variant of `forkIO`:[37]

```
forkFinally :: IO a -> (Either SomeException a -> IO ()) -> IO ThreadId
```

```
forkFinally action fun =
  mask $ \restore ->
    forkIO (do r <- try (restore action); fun r)
```

The `forkFinally` function lets us simplify `async`:

geturlscancel2.hs

```
async :: IO a -> IO (Async a)
async action = do
   m <- newEmptyMVar
   t <- forkFinally action (putMVar m)
   return (Async t m)
```

Now the API is safe. The rule of thumb is that any exception-handling function called as the first thing in a `forkIO` is better written using `forkFinally`. In particular, if you find yourself writing `forkIO (x `finally` y)`, then write `forkFinally x (\_ -> y)` instead. Better still, use the `Async` API, which handles these details for you.[38]

Asynchronous Exceptions: Discussion

This chapter has been full of tricky and subtle details—such is life when dealing with exceptions that can strike at any moment. The abstractions we've covered in this chapter like `timeout` and `Chan` are certainly hard to get right, but it is worth reminding ourselves that dealing with asynchronous exceptions at this level is something that Haskell programmers rarely have to do, for a couple of reasons:

- All non-IO Haskell code is automatically safe by construction. This is the one factor that makes asynchronous exceptions feasible.
- We can use the abstractions provided, such as `bracket`, to acquire and release resources. These abstractions have asynchronous-exception safety built in. Similarly, when working with `MVar`s, the `modifyMVar` family of operations provides built-in safety.

We find that making most `IO` monad code safe is straightforward, but for those cases where things get a bit complicated, a couple of techniques can simplify matters:

- Large chunks of heavily stateful code can be wrapped in a `mask`, which drops into polling mode for asynchronous exceptions. This is much easier to work with. The problem then boils down to finding the interruptible operations and ensuring that exceptions raised by those will not cause problems. The GHC I/O library uses this technique: every `Handle` operation runs entirely inside `mask`.
- Using software transactional memory (STM) instead of `MVar`s or other state representations can sweep away all the complexity in one go. STM allows us to combine multiple operations in a single *atomic* unit, which means we don't have to worry about restoring state if an exception strikes in the

middle. We will describe STM in Chapter 10.

In exchange for asynchronous-exception-safety, Haskell's approach to asynchronous exceptions confers some important benefits:

- Many exceptional conditions map naturally onto asynchronous exceptions. For example, stack overflow and user interrupt (e.g., Ctrl+C at the console) are mapped to asynchronous exceptions in Haskell. Hence, Ctrl+C not only aborts the program but also does so cleanly, running all the exception handlers. Haskell programmers don't have to do anything to enable this behavior.
- Computation can always be interrupted, even if it is third-party library code. (There is an exception to this, namely calls to foreign functions, which we shall discuss in "Threads and Foreign Out-Calls").
- Threads never just die in Haskell. It is guaranteed that a thread always gets a chance to clean up and run its exception handlers.

---

[34] Historical note: the original presentation of asynchronous exceptions used a pair of combinators, `block` and `unblock`, here, but `mask` was introduced in GHC 7.0.1 to provide a more modular behavior and to avoid using the overloaded term "block."

[35] For simplicity here, we are using a slightly less general version of `mask` than the real one in the `Control.Exception` library.

[36] An exception is foreign calls; see "Asynchronous Exceptions and Foreign Calls".

[37] The `forkFinally` function is provided by `Control.Concurrent` from GHC 7.6.1.

[38] The full `Async` library is available in the `async` package on Hackage.
