Live a little: `unsafePerformIO` it!

Victor Miraldo

Channable

Oct 3, 2022

Does this look familiar?

Who has written these towers:

```
type MyFancyT m =
  WriterT MyMsgs (StateT MyState (ReaderT MyEnv (... m)))
```

Does this look familiar?

Who has written these towers:

```
type MyFancyT m =
  WriterT MyMsgs (StateT MyState (ReaderT MyEnv (... m)))
```

Only to hide `IO` at some point...

```
f :: (MonadIO m) => MyFancyT m Integer
```

Does this look familiar?

Who has written these towers:

```
type MyFancyT m =
  WriterT MyMsgs (StateT MyState (ReaderT MyEnv (... m)))
```

Only to hide `IO` at some point...

```
f :: (MonadIO m) => MyFancyT m Integer
```

This is but one technique to cope with `IO`

Today

1. Question Monad-transformer design pattern
2. Explore `unsafePerformIO` as a reasonable alternative

Why did I start thinking of this?

Why did I start thinking of this?

Needed access to an SMT solver. Ideally:

```
solve :: Problem -> Bool
```

Why did I start thinking of this?

Needed access to an SMT solver. Ideally:

```
solve :: Problem -> Bool
```

Which would have a clean and elegant *denotation*:

```
solve x == True iff x is SAT
```

Why did I start thinking of this?

Needed access to an SMT solver. Ideally:

```
solve :: Problem -> Bool
```

Which would have a clean and elegant *denotation*:

```
solve x == True iff x is SAT
```

Yet...

Why did I start thinking of this?

Needed access to an SMT solver. Ideally:

```
solve :: Problem -> Bool
```

Which would have a clean and elegant *denotation*:

```
solve x == True iff x is SAT
```

Yet...

```
solver <- forkIO (exec "z3")
...
```

Stuck in `IO`

And now I was stuck in `IO` with:

```
solve :: Problem -> IO Bool
```

Stuck in `IO`

And now I was stuck in `IO` with:

```
solve :: Problem -> IO Bool
```

`map`s had to become `mapM`: now stuck with *sequential* maps everywhere! (recall `IO` is just glorified state)

Stuck in `IO`

And now I was stuck in `IO` with:

```
solve :: Problem -> IO Bool
```

`map`s had to become `mapM`: now stuck with *sequential* maps everywhere! (recall `IO` is just glorified state)

Sharing opportunities disappeared and I lost laziness:

```
> take 2 <$> mapM print [1..3]
1
2
3
[(), ()]
```

Stuck in `IO`

And now I was stuck in `IO` with:

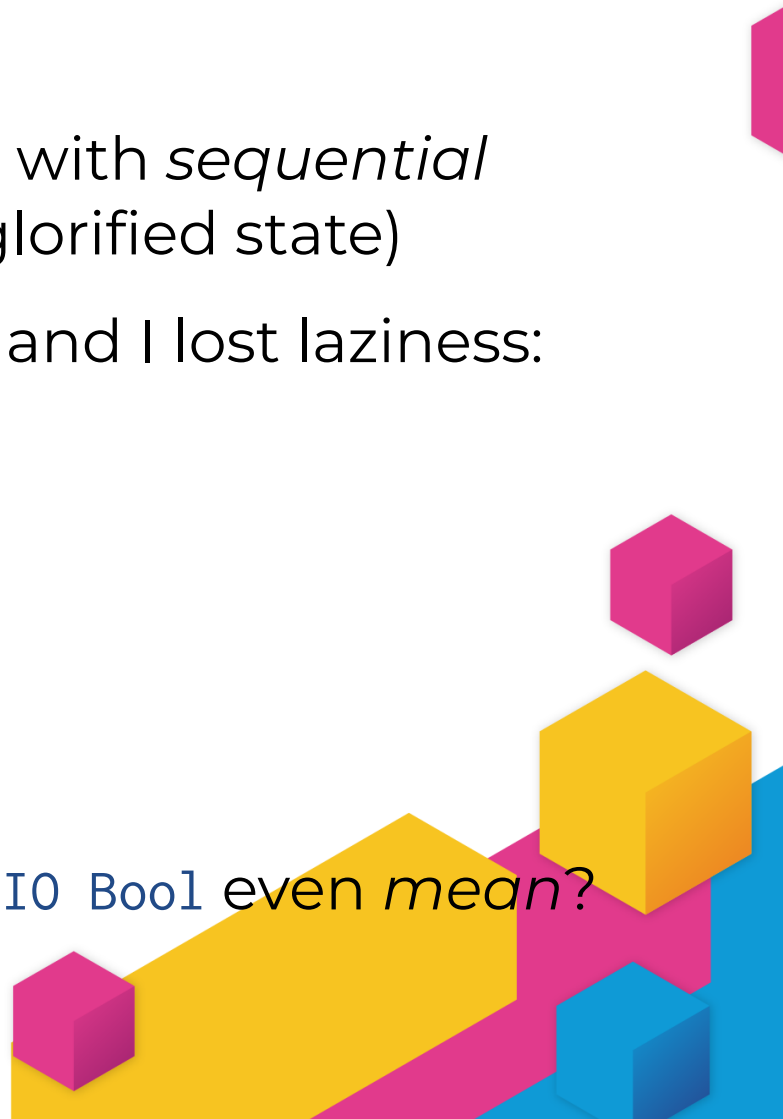
```
solve :: Problem -> IO Bool
```

`map`s had to become `mapM`: now stuck with *sequential* maps everywhere! (recall `IO` is just glorified state)

Sharing opportunities disappeared and I lost laziness:

```
> take 2 <$> mapM print [1..3]
1
2
3
[(), ()]
```

Even worse... What does `solve x :: IO Bool` even *mean*?



Stuck in `IO`

Open challenge 1

Open problem: the IO monad has become Haskell's sin-bin. (Whenever we don't understand something, we toss it in the IO monad.)

Festering sore:

```
unsafePerformIO :: IO a -> a
```

Dangerous, indeed type-unsafe, but occasionally indispensable.

War

Microsoft | Research Our research More Sign up: Research Newsletter All Microsoft

Wearing the hair shirt: a retrospective on Haskell (2003)

Simon Peyton Jones
January 2003
invited talk at POPL 2003

Download BibTex

What did I want?

I wanted (in pseudo-Agda):

```
safePerformIO : (f : IO a) -> ProofOfSafety f -> a
safePerformIO f _ = unsafePerformIO f
```

In particular, to know what `ProofOfSafety` should even look like?

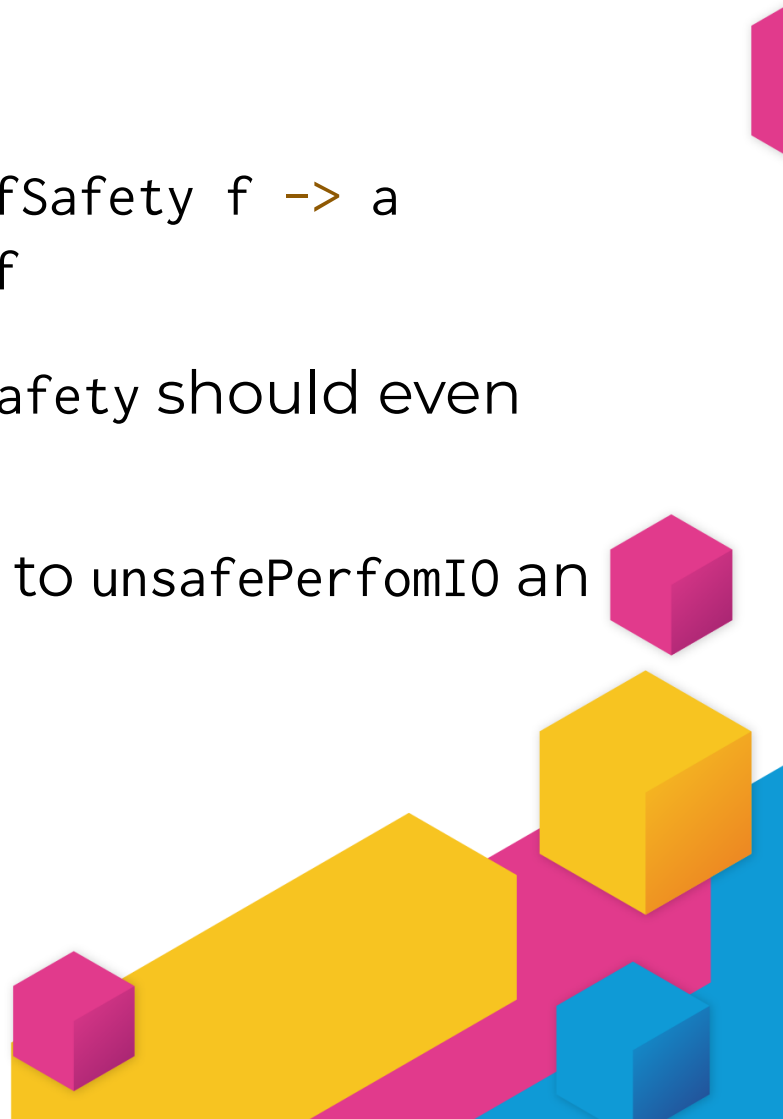
What did I want?

I wanted (in pseudo-Agda):

```
safePerformIO : (f : IO a) -> ProofOfSafety f -> a
safePerformIO f _ = unsafePerformIO f
```

In particular, to know what `ProofOfSafety` should even look like?

That is: how do we conclude it is ok to `unsafePerfomIO` an `IO` block?



Interlude: Some History

Interlude: Some History

At first, Haskell programs were perfectly pure!

```
type Behaviour = [Response] -> [Request]

main :: Behavior
```

Interlude: Some History

At first, Haskell programs were perfectly pure!

```
type Behaviour = [Response] -> [Request]

main :: Behavior
```

In 1996, with Haskell 1.3, we saw monadic IO come up.

```
instance Monad IO where

main :: IO ()
```

Interlude: Some History

At first, Haskell programs were perfectly pure!

```
type Behaviour = [Response] -> [Request]

main :: Behavior
```

In 1996, with Haskell 1.3, we saw monadic IO come up.

```
instance Monad IO where

main :: IO ()
```

For more, check A History of Haskell, section 7.

Are there guidelines for `unsafePerformIO` ?

SPJ gave some nice guidelines back in 2017 (haskell mailing list), there's also a long thread on discourse on this (thanks @rae).

Are there guidelines for `unsafePerformIO` ?

SPJ gave some nice guidelines back in 2017 (haskell mailing list), there's also a long thread on discourse on this (thanks @rae).

TL;DR:

Order is an illusion. You must not care about (or defend against!) the order in which your `unsafePerformIO`s are ran, whether they're ran at all (sequentially or concurrently).

Are there guidelines for `unsafePerformIO` ?

SPJ gave some nice guidelines back in 2017 (haskell mailing list), there's also a long thread on discourse on this (thanks @rae).

TL;DR:

Order is an illusion. You must not care about (or defend against!) the order in which your `unsafePerformIO`s are ran, whether they're ran at all (sequentially or concurrently).

Good necessary rule of thumb:

```
let x = unsafePerformIO f in (x , x)
  <=>
(unsafePerformIO f , unsafePerformIO f)
```

For some domain-specific definition of <=>.

RTFM!

Which was already written in the docs, anyway...

> *If the I/O computation wrapped in* `unsafePerformIO` *performs side effects, then the relative order in which those side effects take place (relative to the main I/O trunk, or other calls to unsafePerformIO) is* ***indeterminate****. Furthermore, when using* `unsafePerformIO` *to cause side-effects, you should take the [...] precautions to ensure the side effects are performed as many times as you expect them to be. [...]*

But can’t it crash?

But can’t it crash?

We already accept a number of failures in pure code anyway:

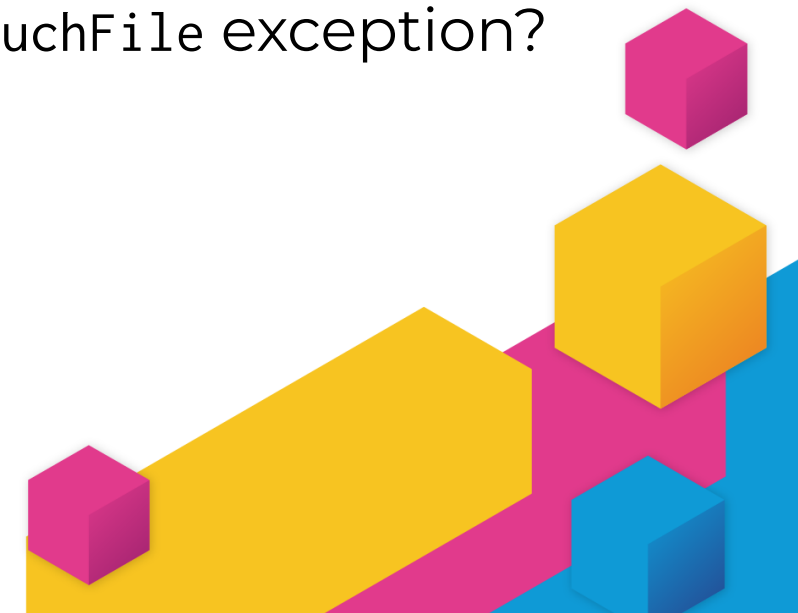
1. Out-of-memory exceptions
2. Async exceptions (i.e., SIGINT)

But can’t it crash?

We already accept a number of failures in pure code anyway:

1. Out-of-memory exceptions
2. Async exceptions (i.e., SIGINT)

What is so much worse about a `NoSuchFile` exception?



Lies everywhere!

There are a number of impure operations we accept as pure:

1. Memory allocation
2. Function calling (can create stack frames and jumps)

TL;DR:

Do not dismiss `unsafePerformIO`, it's a valuable tool to cope with `IO`

TL;DR:

Do not dismiss `unsafePerformIO`, it's a valuable tool to cope with `IO`

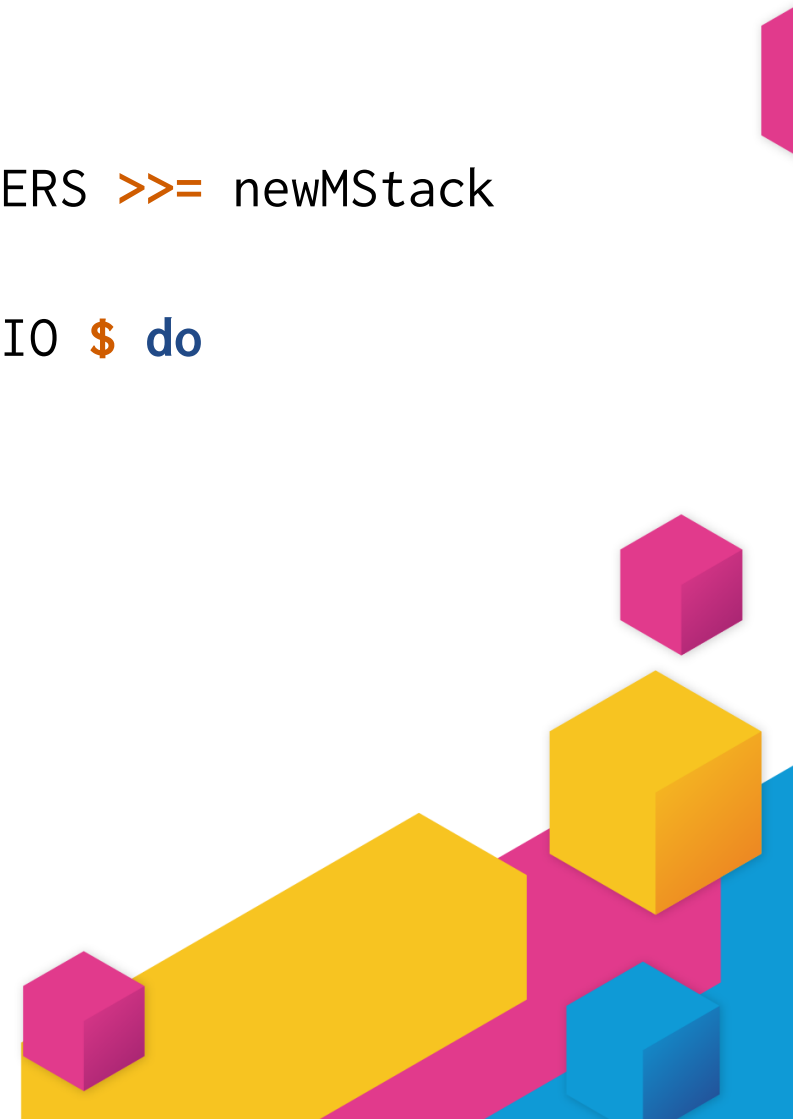
Depending on the context, more valuable than hiding `IO` inside monad transformers.

What did I end up with?

What did I end up with?

```
solve :: Problem -> Bool
solve = unsafePerformIO $ do
  allProcs <- launchSolvers nUM_SOLVERS >>= newMStack

  return $ \problem -> unsafePerformIO $ do
    ms <- popMStack allProcs
    r <- withMVar ms $ \solver -> do
      solveProblem problem solver
    pushMStack ms allProcs
    return r
```



`unsafePerformIO` as an implementation trick

Just like registers, stack frames and jumps are an implementation trick to run our software in the current hardware,

`unsafePerformIO` as an implementation trick

Just like registers, stack frames and jumps are an implementation trick to run our software in the current hardware,

Think of `unsafePerformIO` as an implementation trick: How do we implement our software over the current user-interation API?

`unsafePerformIO` as an implementation trick

Just like registers, stack frames and jumps are an implementation trick to run our software in the current hardware,

Think of `unsafePerformIO` as an implementation trick: How do we implement our software over the current user-interation API?

Should we pollute every user *and every use* of our code with the weight of monads?

Advantages of `unsafePerformIO`:

Advantages of `unsafePerformIO`:

Semantics!

Advantages of `unsafePerformIO`:

Semantics!

```
conc :: A -> B -> ... -> IO Result
conc x y ... = uglyIOcode
```

versus

```
abstr :: A -> B -> ... -> Result
abstr x y ... = unsafePerformIO $ conc x y ...
```

Advantages of `unsafePerformIO`:

Semantics!

```
conc :: A -> B -> ... -> IO Result
conc x y ... = uglyIOcode
```

versus

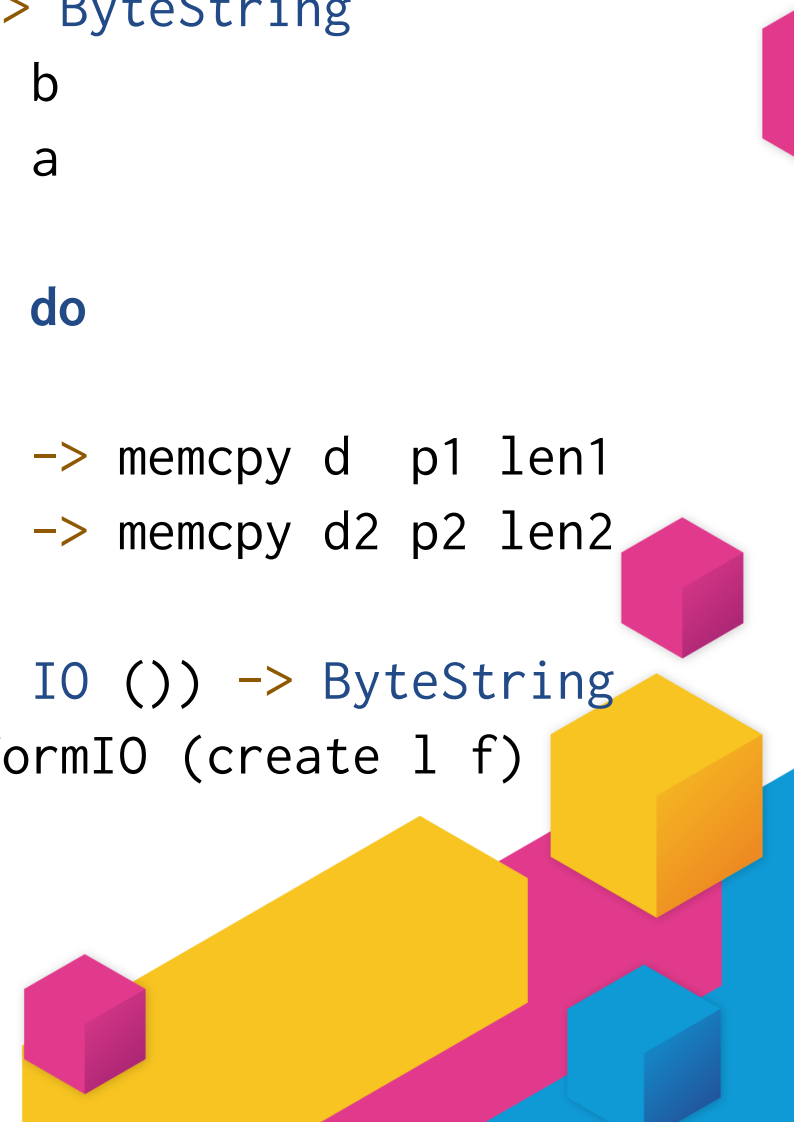
```
abstr :: A -> B -> ... -> Result
abstr x y ... = unsafePerformIO $ conc x y ...
```

`conc` is just one possible implementation for what `abstr` is *supposed to be*.

Example: bytestring

```
append :: ByteString -> ByteString -> ByteString
append (BS _   0)    b                 = b
append a             (BS _   0)        = a
append (BS fp1 len1) (BS fp2 len2) =
    unsafeCreate (len1+len2) $ \d -> do
      let d2 = d `plusPtr` len1
      unsafeWithForeignPtr fp1 $ \p1 -> memcpy d  p1 len1
      unsafeWithForeignPtr fp2 $ \p2 -> memcpy d2 p2 len2

unsafeCreate :: Int -> (Ptr Word8 -> IO ()) -> ByteString
unsafeCreate l f = unsafeDupablePerformIO (create l f)
```



Example: `bytestring`

`append` denotes something and should satisfy laws!

Example: `bytestring`

`append` denotes something and should satisfy laws!

Different implementations must satisfy *the same laws*

Wrapping it up

1. `unsafePerformIO` shouldn't be immediately dismissed as some sort of boogeyman;

Wrapping it up

1. `unsafePerformIO` shouldn't be immediately dismissed as some sort of boogeyman;

2. Thinking in terms of meanings is more important than in terms of implementations;

Wrapping it up

1. `unsafePerformIO` shouldn't be immediately dismissed as some sort of boogeyman;
2. Thinking in terms of meanings is more important than in terms of implementations;
3. Out of the 50 most used libraries on Hackage, 35 use `unsafePerformIO`;

Wrapping it up

1. `unsafePerformIO` shouldn't be immediately dismissed as some sort of boogeyman;
2. Thinking in terms of meanings is more important than in terms of implementations;
3. Out of the 50 most used libraries on Hackage, 35 use `unsafePerformIO`;
4. Purity is really difficult, if not impossible, to define: heavily contextual