

Principled, Painless Asynchronous Programming in PureScript

John A. De Goes — [@jdegoes](#)

Callback Hell¹

```
getData(a, function(b) {  
  getData(b, function(c) {  
    getData(c, function(d) {  
      getData(d, function(e) {  
        getData(e, function(f) {  
          .  
          .  
          .  
        });  
      });  
    });  
  });  
});
```

¹ That's *without* error handling!

```
then(onFulfilled, onRejected)
```

`onFulfilled` and `onRejected` are optional arguments:

`onFulfilled` is not a function, it must be ignored.

`onRejected` is not a function, it must be ignored.

`onFulfilled` is a function:

Must be called after `promise` is fulfilled, with `promise` 's value as its first argument.

Must not be called before `promise` is fulfilled.

Must not be called more than once.

`onRejected` is a function,

Must be called after `promise` is rejected, with `promise` 's reason as its first argument.

Must not be called before `promise` is rejected.

Must not be called more than once.

`onFulfilled` or `onRejected` must not be called until the [execution context](#) stack contains only one frame.

code. [3.1].

`onFulfilled` and `onRejected` must be called as functions (i.e. with no `this` value).

May be called multiple times on the same promise.

When `promise` is fulfilled, all respective `onFulfilled` callbacks must execute their originating calls to `then`.

When `promise` is rejected, all respective `onRejected` callbacks must execute their originating calls to `then`.

Must return a promise [3.3].

```
promise2 = promise1.then(onFulfilled, onRejected);
```

Whether `onFulfilled` or `onRejected` returns a value `x`, run the Promise Resolution Procedure `[[Resolve]](promise2, x)`.

Whether `onFulfilled` or `onRejected` throws an exception `e`, `promise2` must be rejected with `e` as the reason.

If `onFulfilled` is not a function and `promise1` is fulfilled, `promise2` must be fulfilled with the same value as `promise1`.

If `onRejected` is not a function and `promise1` is rejected, `promise2` must be rejected with the same reason as `promise1`.

CancellationException

A promise is *directly cancelled* if it is rejected with a `CancellationException`. A `CancellationException` must have the following points:

1. It must be an instance of `Error` (`cancellationException instanceof Error === true`).

2. It must have a property `.name = 'OperationCancelled'`.

3. It must have a property `.cancelled = true`.

cancel Method

The `cancel` method is called on a promise it is *directly cancelled*. The `cancel` method accepts two optional arguments:

```
.cancel(reason, data);
```

A promise **MUST** be rejected with a `CancellationException`.

`reason` and `data` are optional.

If `reason` is not `undefined` it **SHOULD** default to `OperationCancelled`.

If `data` is `undefined` it **SHOULD** be ignored.

If `reason` is a string it is used as the `message` property in the `CancellationException`.

If `data` is not `undefined` it is set as the `data` property of the `CancellationException`.

A promise must call `this.propagateCancel()` and return the result.

propagateCancel method

The `propagateCancel` method is intended only to be called by this and other promises, it is not for use by application code.

When

1. `propagateCancel` is called, the promise transitions into an extra `cancelled` state. This does not affect the state of the promise.

2. It does however mean that the promise can never be resolved (i.e. it never leaves the `pending` state).

When

1. A promise is waiting on another promise to complete it **may** call `.propagateCancel()` on the promise it is waiting for.

2. It must return a promise for the result of calling any cancellation handlers attached to the promise it is waiting for.

3. If the promise it is waiting for rejects it should call `.propagateCancel()` on the promise it's waiting for. The exception is if the promise it is waiting for has been in some way 'forked', when it may choose not to in an implementation specific manner.

Uncaught Promise Rejection

A "handled rejection": the occurrence wherein a promise transitions to the rejected state, but there is at least one rejection handler registered for it.

A "unhandled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

Uncaught Promise Rejection

A "handled rejection": the occurrence wherein a promise transitions to the rejected state, but there is at least one rejection handler registered for it.

A "unhandled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

Uncaught Promise Rejection

A "handled rejection": the occurrence wherein a promise transitions to the rejected state, but there is at least one rejection handler registered for it.

A "unhandled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

A "handled rejection": if a promise is in the rejection state with no rejection handlers, but then one is added later.

What about Promises/A+?

Introducing Aff²

do

```
b <- getData a
```

```
c <- getData b
```

```
d <- getData c
```

```
e <- getData d
```

```
.
```

```
.
```

```
.
```

² That's with error handling *baked in*!

purescript-aff

- Asynchronous programming shouldn't look any different than synchronous programming (`Eff` / `Aff`).
- Asynchronous primitives should be few in number, low-level, and have principled semantics.
- Errant code should not take down the whole application.
- Performance should be very good, better than alternative Javascript libraries.

`Aff eff a`

An asynchronous computation with effects
'`eff`' that will yield a value of type '`a`' or
terminate with an '`Error`'.

Under the Hood

```
ajaxGet :: forall eff. String -> Aff (ajax :: AJAX | eff) Response
```

```
function ajaxGet(url) {  
  return function(success, failure) { // <- Aff!  
    ...  
    return canceler;  
  };  
}
```

Small but Rich API

- **13 Instances:** Semigroup, Monoid, Functor, Apply, Applicative, Bind, Monad, MonadEff, MonadError, Alt, Plus, Alternative, MonadPlus.
- **9 Functions:** runAff, launchAff, makeAff, forkAff, attempt, apathize, later, later', finally.

Pure Aff

pure 42

Sequential Composition

It's the same as `Eff`: looks & acts like synchronous code.³

`do`

```
b <- Ajax.get a
```

```
c <- Ajax.get b
```

```
...
```

³ See also, Haskell's `IO`. `Sync/async` is an implementation detail!

Delayed Computation

Delaying a computation does not add effects, since the fact the computation is asynchronous is already captured in `Aff`.

do

```
a <- later          $ pure 42
b <- later' 1000    $ pure 58
return $ a + b
```

Running an Aff

To *run* an `Aff`, you must supply callbacks to handle success and failure cases.

```
main = do
  runAff (const $ trace "Oh noes!")          -- failure
        (\v -> trace "Got value: " ++ v)    -- success
        (pure 42)
```

Launching an Aff

To *launch* an `Aff`, you needn't supply anything, but both value and errors will be discarded.

```
main = do  
  launchAff $ pure 42
```

Converting from Callbacks

```
type Ajax eff = (ajax :: AJAX | eff)

ajaxGet0 :: forall eff.
  String -- url
  (Error -> Eff (Ajax eff) Unit) -> -- error callback
  (Response -> Eff (Ajax eff) Unit) -> -- success callback
  Eff (Ajax eff) Unit

ajaxGet :: forall eff. String -> Aff (Ajax eff) Response
ajaxGet url = makeAff $ ajaxGet0 url
```

Converting from Eff

Lifted `Eff` computations are run immediately; `Aff` is a *strict* superset of `Eff`.

```
main = do
  liftEff $ trace "hello world!"
```

Errors

```
attempt :: forall eff a. Aff eff a -> Aff eff (Either Error a)
```

```
method1 = do
  either <- attempt doX
  case either of
    Left  err -> doY
    Right val -> pure val
```

```
method2 = doX <|> doY -- if first fails, tries second
```

```
method3 = catchError (throwError $ error "Oh noes!") (const $ pure 42)
```


Forking

Forking is *like* spawning a separate thread for computation.⁴

do

```
forkAff $ later (trace "makes jack a dull boy")
liftEff $ trace "All work and no play"
```

⁴ Except, of course, Javascript has no threads, but the effect (affect?) is the same. 😊

Killing a Forked Computation

```
do  
  c <- forkAff $ later (trace "You'll never see this")  
  _ <- cancel c (error $ "Cause")  
  liftEff $ trace "But you will see this!"
```

AVar

AVar lets you communicate between 'threads'.

do

```
v <- makeVar
```

```
forkAff (later $ putVar v 1.0)
```

```
a <- takeVar v
```

```
trace ("Success: Value " ++ show a)
```

Par

Par lets you trade Bind for parallel composition.

do

[illegible]

```
first    <-  runPar  (Par (later' 100 $ pure "First") <|>
                    Par (later' 200 $ pure "Second"))
```

AStream

Coming Soon

- Asynchronous streams that compose in parallel
 - Effectful or pure sources
 - drop, take, zip, etc.

Aff Ecosystem

A growing number of libraries & projects use Aff.

- [purescript-affjax](#)
- [purescript-halogen](#)
- [purescript-wai](#)
- [purescript-rx](#)
- [purescript-any-db](#)
- [purescript-node-postgres](#)
- [purescript-node-mongodb](#)
- [purescript-routing](#)
- [purescript-spec](#)
- [giflib-web](#)
- [gulp-purescript](#)
- [purefn/naggy](#)

YOUR LIBRARY HERE

THANK YOU

Follow me @jdegoes