## 1 Introduction

One widely used model of concurrency is Javascript's Promises. A promise works in some ways like a lazy value, in that it will at some point will contain the result of a computation, but does not stop flow of control in the current thread to compute that value. Unlike a value with lazy semantics, a promise can immediately begin computation in a separate thread as opposed to waiting for the result to be requested by some other computation.

Promises are composable: using then and catch, we can chain a promise onto the end of a different one creating a new promise that continues computation after the first has succeeded or failed respectively.

We build a model of promises in Haskell to compare it against other concurrency frameworks.

## 2 Motivation

[Discussion of the adoption of the promises model in Javascript; will have to see what I can find citations for.]

Promise objects are a good candidate for parameterized types: they yield a value upon success or failure, and it would be nice to be able to check statically that these values and the functions that they will be passed to all agree on the types involved.

The operation then should "chain promises together" by accepting a function that converts a regular value to a promise, then applying it to a promise with that return type by waiting for it to finish before calling the function. We note that this operation is very similar to >>= [give type signature here?], suggesting that promises can be thought of as monads. (We also note that it is trivial to wrap an arbitrary value into a promise so return poses no problem. In practice, we define a separate constructor for this case to avoid forking off an entire new thread to do no computation and hand back the same result immediately.)

## 3 Background

[HS type signature syntax - mention here that I'm using ? in type signatures I'm discussing as a metavariable, not real syntax?] [IO monad, do notation] [forkIO, description of MVar] [maybe discuss GADTs briefly ?]

[Terminology for discussing promises: states *pending, fulfilled, rejected*; resolve means pending -> fulfilled, reject - pending -> rejected, settle - pending -> <either>; define promise chain; Promises here ≠ lazy values, despite terminology clash w/ e.g. Scheme]

## 4 Haskell Implementation

[core bits] Javascript's Promise() constructor builds a new promise object from an 'executor' function. The executor accecpts two callback functions the standard names resolutionFunc and rejectionFunc, one to call in the case of successful resolution and the other for failure. The executor will, on a success/failure, call resolutionFunc/rejectionFunc, repectively, passing in the value of or reason for the success or failure. Let's assume we want a promise with type `Promise f p`, i.e. one where success results in a value of type p and failure gives a reason with type f. To build one, `resolutionFunc` will need to accept a value of type p. Since calling `resolutionFunc` will settle the promise and therefore have effects elsewhere in the promise chain, its return type will have to be something wrapped in IO, so we know `resolutionFunc :: p -> IO ?`. Similarly, `rejectionFunc` must accept a f, and calling it will also settle the promise, so `rejectionFunc :: f -> IO ?`. The executor function should accept `resolutionFunc` and `rejectionFunc` as parameters and is expected to end by calling exactly one of them, so we will expect it to have a proper tail call to one of the parameters. This means its return type matches that of `resolutionFunc` and `rejectionFunc`, i.e. `executor :: (p -> IO ?) -> (p -> IO ?) -> IO ?` and the ?s for the two callbacks should be the same type. [For now, let's use `()` as ?, so that the callbacks have a return type of `IO ()`, the conventional Haskell type for IO actions that have an effect (here, setting the Promise from Pending state to one of the settled states) instead of containing a useful value.] Our function for building a Promise object needs to accept a function with the type of executor and give back a Promise value, which must be contained in IO because it has the side effect of running the executor in another thread. It thus has the type `newPromise :: ((p -> IO ()) -> (f -> IO ()) -> IO ()) -> IO (Promise f p)`. We can represent a `Promise f p` by an MVar `(Either f p)`. Once the computation for the Promise is complete, it can be written to with an `Either f p` value, i.e. `Left` reason for a failure or `Right` result in the case of success. `newPromise` will also need to fork a thread that will run the executor and set up communication so that the final Promise object will be updated with the results once they are available. In total, we need to: create an MVar which we'll call `state`, then fork a thread that calls the executor, passing it callback functions that write the results to `state`, and finally, return `state` as a Promise value.

```
newPromise :: ((p -> IO ()) -> (f -> IO ()) -> IO ()) -> IO (Promise f p)
newPromise k = do
  state <- newEmptyMVar
  forkIO $ k (putMVar state . Right) (putMVar state . Left)
  return (Pending state)
```

Since the constructor here is used to create Promises that are in the *pending* state, we'll call it Pending. We could, in principle, use this same constructor to build Promise values that we know have already succeeded or failed. To get a promise that always succeeds with a value of s, say, simply call newPromise with an executor that immediately calls successFunc, like so:

```
newPromise (\ succeed fail -> succeed s)
```

This is inefficient, though, because it spawns an entire new thread in order to do absolutely nothing with it. Instead, it is easy enough to define a constructor that marks a value as known to be the result of a sucessful computation (and a parallel one declaring a value to be the known reason for a failed computation). These correspond to the promise being in the state *fulfilled* or *rejected*, repectively, so we will uses those terms as the names the constructors. At this point, the Promise type has the following form [(in GADT syntax)]:

```
data Promise :: * -> * -> * where
  Pending :: MVar (Either f p) -> Promise f p
  Fulfilled :: p -> Promise f p
  Rejected :: f -> Promise f p
```

[side note: could use Void as ? instead of () - this would cause the compiler to check that at least one of the callbacks is used - reworked into Token type with non exported constuctor; maybe provide hangForever :: Token (or IO Token)so it can be used but can't be accidental?]

[then and catch] Now that we can create Promise values, the next step is to allow them to chain together. Javascript's Promise.then() is used to set a handler function to run after a promise completes. Specifically, p1.then(f) results in a new promise that will wait for the Promise p1 to complete. If p1 succeeds and resolves to a value v, it will then call f(v). The result of running the callback should be another Promise, p2; when it settles, the new Promise will also settle, to the same state and value. [JS also allows the callback to return a non-Promise value, in which case p1.then(f) resolves to that value as soon as it's computed. We won't implement this functionality directly (allowing differeng argument types would not work in Haskell's type system unless we made separate then functions for the two variants). However, we get to the same result by enclosing the value we would like to return in an always-successful Promise. Once we define a monad instance for Promise f, we can even do so by writing return v where v is the value for the final promise to resolve to, which should look familiar to anyone used to the Javascript syntax!] In our system in Haskell, pThen accepts pr, a Promise f p along with a callback that expects a value type p, the type contained in a sucessful Promise f p. pThen will return a Promise (in IO because [blah blah asynchronous], which must have the same failure type as pr because if pr is Rejected, the result will be as well, with the same value. The result type can have a different sucess type, though, so it's overall type is IO (Promise f p'). The callback returns a new Promise in IO, which must match pThen's return type, so in total

```
pThen :: Promise f p
        -> (p -> IO (Promise f p'))
        -> IO (Promise f p')
```

[implementation without runPromise, for only the Pending, Fulfilled, Rejected constructors.]

```
pThen (Pending state) k = do
  result <- readMVar state
  case result of
    Left x -> return $ reject x
    Right x -> k x
pThen (Fulfilled x) k = k x
pThen (Rejected x) k = return $ reject x
```

[Note that the type signature for pThen looks extremely similar to the type (>>=) would have if it were to be specialized to Promise f ((>>=) :: Promise f a -> (a -> Promise f b) -> Promise f b). The difference is that pThen is entangled in the IO monad.] Promise.catch() works the same way as .then() except that the handler is set to run only if and when the Promise it is being chained to fails, rather then when it succeeds. Our translation to Haskell, pCatch, is very much like pThen except that the code for a failed promise and a successful one have swapped places. Its type is

```
pCatch :: Promise f p
        -> (f -> IO (Promise f' p))
        -> IO (Promise f' p)
```

which is the same as that for pThen except that it operates on the type f, the type of failure cases, instead of p, the type of success cases. [Implementation. ]

```
pCatch (Pending state) k = do
  result <- readMVar state
  case result of
    Left x -> k x
    Right x -> return $ resolve x
pCatch (Fulfilled x) k = return $ resolve x
pCatch (Rejected x) k = k x
```

pCatch is dual to pThen in that it is identical to a pThen that operates on Promises with reversed semantics for which type argument represents success and which failure.

[runpromise as helper function] pThen and pCatch both share the same central function of waiting, if necessary, for a Promise to settle, then branching on whether the result was a success or a failure. We can generalize this behavior by writing a single function that accepts arguments specifying what to do in either case. The action to do in the case of success can depend on the particular value the promise resolved with, so it should be a function accepting values of type p.[Name the parameter - code calls it yes] The overall result [of runPromise] must be contained in the IO monad because we can only compute it with the side effect of waiting for the Promise[this would flow easier if I named the parameter] to settle. [yes should match the runPromise return value so :: p -> IO ?, no more restrictions on ? => :: p -> IO a ; no must match return types therefore :: f -> IO a]

```
runPromise :: (p -> IO a) -> (f -> IO a) -> Promise f p -> IO a
runPromise yes no (Pending state) = do
  result <- readMVar state
  case result of
    Left x -> no x
    Right x -> yes x
runPromise yes _ (Fulfilled x) = yes x
runPromise _ no (Rejected x) = no x
```

[reimplement pThen, pCatch with runPromise] [runPromise has the semantics of the two-arg form of JS then]

[ pFinally ]

[ Functor, Applicative, Monad instances; addition to the type so they don't need to be in IO] [ Proofs of typeclass laws (maybe put this elsewhere?) ]

The Javascript standard library has several ways to combine promises in parallel in addition to the sequential combination provided by then and catch.

[first describe implementation for pRace2 - it's simpler] [JS provides Promise.race( iterable )], which runs all of the input promises simultaneously in different threads, settling with the result of whichever completes first. In our system this should have type signature pRace :: [Promise f p] -> IO (Promise f p). To implement this function, let's begin with a binary variant that works for exactly two promises. pRace2 :: Promise f p -> Promise f p -> IO (Promise f p). [similar to amb from [cite Push-Pull FRP]] We can use an MVar to accept a result from the first thread to finish. Since we must differentiate between whether the result is a success or failure, we want the MVar to hold an Either f p. We create an empty MVar, then fork off a pair of threads, each of which runs one of the input promises and writes the result to the MVar. Next, takeMVar waits for either thread to finish and give it a result, after which we can kill both threads since they are no longer needed. [insert pRace2 code here]

The *n*-ary version of pRace operates by a sort of monadic fold over the list of input promises: we pRace2 the first promise in the list against the result of pRaceing the rest of the list, with the result that we will settle to whichever out of any of the inputs settles first. The Javascript standard specifies that race()ing an empty iterable returns a forever-pending promise that never resolves or rejects. This is convenient for our implementation because such a promise is the identity for pRace2 so we can use it directly as the base case to our fold. We can generate an eternally pending promise by passing newPromise a function that fails to call either the success or failure handle, like so: `newPromise (\s f -> return ())`, so the final pRace function is as follows: [...]

[implementation of JS's Promise.any(iterable)] Javascript's promise API also provides .any( iterable ) which combines any number of promises by executing each simultaneously. The result is a promise that immediately resolves to the value of the first input promise to successfully complete. If all of the given promises fail, it gives a list of every failure value. To implement this, let's again start with a binary version that combines exactly two promises in this way. [ type signature pAny2 :: Promise f p -> Promise f' p -> IO (Promise (f, f') p) ] [Note that this type signature is slightly more general

than will be allowed by the *n*-ary version; in particular, the failure types of the two Promises can be different here, where in pAny they will need to be the same so they can be contained in the same Haskell list.] We still need an MVar to store the value of a success from either promise A or promise B, but dealing with a failure is somewhat more complicated since one failure isn't enough to end the computation, but we still need to track it so that we know to end if both branches end in failure. [need communication between the forked threads that doesn't interfere with the MVar holding successful results, => second MVar that the main thread doesn't touch at all. One fork writes to the error MVar, while the other waits to read from it after completion.]

[pAll2 and pAll are dual to pAny2 and pAny; we can implement by using PromiseInvert to switch the true and false cases of the input promises, then switching back after running them through the dual function.]

## 5 Comparison with Existing Concurrency Frameworks

[Compare to push-pull frp] [locks]

## 6 Future Work

[decouple Promise monad from IO ?]