Promise Land
Proving Correctness with Strongly Typed Javascript-Style Promises

Andrei Elliott

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Committee members

Matt Moore, Chair

Perry Alexander

Drew Davidson

Date defended: Friday May 6, 2022

The Project Report Committee for Andrei Elliott certifies
that this is the approved version of the following project report :

Promise Land
Proving Correctness with Strongly Typed Javascript-Style Promises

Matt Moore, Chair

Date approved: -

Abstract

Code that can run asynchronously is important in a wide variety of situations, from user interfaces to communication over networks, to the use of concurrency for performance gains. One widely-used method of specifying asynchronous control flow is the `Promise` model as used in Javascript. `Promises` are powerful, but can be confusing and hard-to-debug. This problem is exacerbated by Javascript's permissive type system, where erroneous code is likely to fail silently, with values being implicitly coerced into unexpected types at runtime.

The present work implements Javascript-style `Promises` in Haskell, translating the model to a strongly typed framework where we can use the type system to rule out some classes of bugs. Common errors – such as failure to call one of the callbacks of an executor, which would, in Javascript, leave the `Promise` in an eternally-*pending* deadlock state – can be detected for free by the type system at compile time and corrected without even needing to run the code.

We also demonstrate that `Promises` form a monad, providing a monad instance that allows code using `Promises` to be written using Haskell's *do notation*.

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 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.

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. Note that this operation is very similar to $\gg=$ [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.)

Background

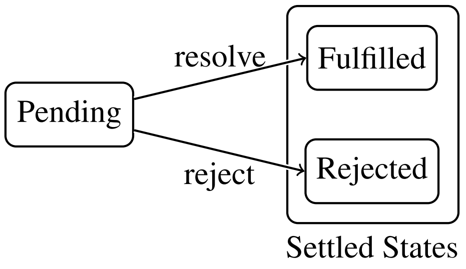
Haskell

[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 ?]

Promises

[Promises here $\neq$ lazy values, despite terminology clash w/ e.g. Scheme] `Promises` can be in one of three states. The *pending* state represents a `Promise` that is still running. A `Promise` that has completed with a success value is in the *fulfilled* state; the process of moving from *pending* to *fulfilled* is referred to as the `Promise` *resolving*. The state for a failed `Promise` is called *rejected* and to *reject* a `Promise` is to move it from the *pending* state to the *rejected* state. For a `Promise` to *settle*, it moves from *pending* to either *fulfilled* or *rejected*. Figure 1 summarizes this terminology.

Figure 1: states and transitions



Comparison with Existing Concurrency Frameworks

[Compare to push-pull frp] [locks]

Haskell Implementation

Making a Promise

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 only 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
```

What happens if we use a type other than `()` in place of `?` in the `newPromise` function? Say we use the type τ. The executor function passed in must evaluate to an `IO` τ. If the executor ends in a call to either `resolutionFunc` or `rejectionFunc`, it will work exactly the same no matter what type τ represents. When using `newPromise`, we can use the same code we did before in the $\tau =$ `()` case. [When `executor` doesn't contain a tail-call to one of its argument functions, the type τ matters for whether `newPromise executor` typechecks; in particular, it will be accepted if and only if whatever `executor` is doing *other than* calling one of its callbacks yields the result type `IO` τ. In this situation, the resulting `Promise` will never settle and any further actions chained to it will never run. Unintentionally causing this state of affairs in that manner is a [common (?check this)] cause of errors in javascript code using `Promises`. [cite] If we were to select a type τ that doesn't appear as the result of normal code, we could have Haskell's type system automatically detect this entire class of bugs at compile time. One option would be Haskell's `Void` type, which has no constructors. But there may be cases where we legitimately need a `Promise` that will never resolve (for example, the Javascript standard specifes that the result of calling `Promise.race()` on the empty list results in such a `Promise`). Furthermore, we need to create a value of type `IO` τ when implementing `newPromise`. Therefore, instead of using `Void`, we create a new type unused anywhere else. [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.] We can provide a value `hangForever :: IO Token` representing the behavior of remaining in the *pending* state indefinitely and never resolving. This allows the user who wants that behavior to specify it while making it unlikely to occur by accident.

What Next?

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. [Javascript 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 of 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
```

Now we can avoid code duplication by rewriting pThen and pCatch in terms of runPromise, as follows:

```
pThen p k = runPromise k (return . reject) p


pCatch p k = runPromise (return . resolve) k p
```

[runPromise has the semantics of the two-arg form of JS then]

Similarly to Javascript's `Promise.finally()`, the function `pFinally` runs a `Promise`, then chains to the `Promise` passed as its argument regardless of how the former settles. We can implement it by generating the function `const k` which ignores its input and always returns `k`, the `Promise` to chain to. We then pass this constant function as both the `yes` and `no` arguments to `runPromise`.

```
pFinally :: Promise f p
         -> IO (Promise f' p')
         -> IO (Promise f' p')
pFinally p k = runPromise (const k) (const k) p
```

[`await`; definition, why it's useful, why use the name]

[instances]

[Functor, Applicative, Monad instances; addition to the type so they don't need to be in IO] [explanation for why partially applied instances (`Promise f`)] To define a Monad instance for `Promise f`, we begin with `Functor` and `Applicative` instances. For `Functor (Promise f)`, we must define `fmap` with type `(a -> b) -> Promise f a -> Promise f b`. `fmap` must accept a function, `g`, and a `Promise`, `pr`, as input and apply the `g` to the success value of `pr` if there is such a value, to yield a new `Promise` (`fmap` will have no effect on a `Promise` that fails; we wouldn't be able to apply `g` to the failure value since it has the wrong type). It is simple enough to run `pr` and then either apply `g` to the result on a success or not on a failure, like so:

```
  fmap' :: (a - > b) -> Promise f a -> IO (Promise f b)
  fmap' g pr = runPromise (return . resolve . g) (return . reject) pr
```

But we have a problem: computing `fmap'` has a side effect - it waits until `pr` has settled. This side effect shows up in the type as we can see that `fmap'` generates an `IO (Promise f b)` instead of a `Promise f b`. To declare a `Functor` instance, the type of `fmap` is specified exaclty. `fmap'` isn't good enough - `Functors` can be mapped over anywhere, not just inside the IO monad. What we can do instead is store g, so we can wait to apply it until we *are* instructed to run `pr`. We can store g by defining another constructor for `Promise f p`. We now know that there is another way to make a `Promise` object: take an existing `Promise` and store along with it a function to map over it. We add a new line to the `Promise` GADT, which now reads:

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

At this point declaring the instance is as simple as telling Haskell to convert `fmap` to our `PromiseMap` constructor:

```
instance Functor (Promise f) where
    fmap g pr = PromiseMap g pr
```

[If that seemed too easy, that's because it was; we still don't have anything like `fmap'` to actually apply `g` when it needs to be applied.] Our definition for `runPromise` needs to say what to do when we try to run a `PromiseMap`. [The pattern matching here looks like `runPromise yes no (PromiseMap g pr)`]Unlike when defining `fmap`, at this point, we are returning an `IO Promise` so we can wait for the contained promise `pr` to settle and decide whether or not to apply g. We can make a recursive call to `runPromise` on `pr`; we know this will terminate becuase `pr` is structurally smaller than `PromiseMap g pr` [, so we're fine unless we're trying to run a `Promise` with an infinite number of functions mapped over it]. The `no` function is unchanged since mapping over a failed `Promise` has no effect, but in the case of a successful one, we need to call `g` before we give the result to `yes`. This means the success function for the recursive call will be `yes .` g, the composition of yes and g, that applies g, then gives the result directly to yes.

```
runPromise yes no (PromiseMap g pr) = runPromise (yes . g) no pr
```

To define the Applicative instance `Applicative (Promise f)`, we need to be able to put an arbitrary value into a `Promise f` and to map a function that is itself the result of a `Promise f` over the (successful) result of another `Promise f`. The first function we must provide is `pure :: a -> Promise f a`. `pure` should put its argument into the context of a `Promise` 'containing' nothing else, which is precisely what `resolve` does. The other function to define for the Applicative instance is `(<*>) :: Promise f (a -> b) -> Promise f a -> Promise f b`, which is like `fmap` except that the function is also inside a `Promise`. Directly running the `Promises` to get their results to combine can't happen outside `IO`, so we will again need to encode the map into a new constructor for `Promise` and unpack it in `runPromise` to avoid the extraneous `IO` in the type. Rather than encoding `(<*>)` directly, we can instead use the equivalent `liftA2` construction that maps a two-argument function over two instances of the applicative. Specialized to `Applicative (Promise f)`, `liftA2` has the type `(a -> b -> c) -> Promise f a -> Promise f c -> Promise f c`. Given such a function, we can implement `(<*>)` as `f <*> x = liftA2 ($) f x`, where `($)` is the application function that accepts a function and an argument and applies one to the other. Our new constructor is called `PromiseMap2` because it maps over two arguments, and we add it to the GADT for `Promiise`:

```
  PromiseMap2 :: (a -> b -> c) -> Promise f a -> Promise f b -> Promise f c
```

and we define the instance as follows:

```
instance Applicative (Promise f) where
  pure x = resolve x
  f <*> x = PromiseMap2 ($) f x
```

The new case to `runPromise` for mapping a function g across two `Promises` creates a `Promise` chain that waits for both arguments to resolve, then yields the value of g applied to the results.

```
runPromise yes no (PromiseMap2 g prA prB) = do
  pr' <- pThen prA $ \a ->
```

```
    pThen prB $ \b -> return $ resolve $ g a b
  runPromise yes no pr'
```

Note that, while using a do block here may look circular since we haven't yet defined the monad instance for `Promise f`, this do is in the `IO` monad rather than `Promise`.

[Monad instance] [`join` characterization instead of »=: collapse a two-layer `Promise` into a single layer]

[Proofs of typeclass laws (maybe put this elsewhere?)]

[parallel combiners]

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

The simplest of the parallel combiners is `Promise.allSettled(iterable)`, which combines all of its input promises into a single `Promise` that runs them in parallel and resolves to a list of each individual result once they are all complete. In Haskell, we can implement `pAllSettled :: [Promise f p] -> IO (Promise f' [Either f p])`. This function accepts a list of `Promise f p` (these `Promises` must have the same success and failure types to fit into a Haskell list) and results in an `IO Promise` object containing a list of the results of each of the `Promises` from the input list. The failure type of the resulting `Promise` is unconstrained because the result of `pAllSettled` is guaranteed to succeed; even if every individual input fails, the result will be a (successful) list of each of the failures. We implement this function recursively as follows. Combining an empty list yields a `Promise` that immediately resolves to the empty list. Otherwise, we run the first list element in parallel with recursing. To do so, we first create an `MVar` for cross-thread communication, then we fork off a thread to `await` the result of the first element and write that to the `MVar`. Next, we recurse, getting a `Promise` holding the results of each of the `Promises` from the tail of the list. At this point, we can read the `MVar`, which will block until the other thread has written to it. Finally, we combine the results into a single promise using `pThen`. Since the

`Promise` we are `pThen`ing to is the result of a call to `pAllSettled`, it is guaranteed to succeed so our code to prepend the new result will always run.

```
pAllSettled :: [Promise f p] -> IO (Promise f' [Either f p])
pAllSettled [] = return $ resolve []
pAllSettled (x:xs) = do v <- newEmptyMVar
                        forkIO $ await x >>= putMVar v
                        prs <- pAllSettled xs
                        a <- takeMVar v
                        pThen prs $ return . resolve . (a:)
```

Javascript also 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.

```
pRace2 :: Promise f p -> Promise f p -> IO (Promise f p)
pRace2 prA prB = do v <- newEmptyMVar
                    ta <- forkIO $ await prA >>= putMVar v
                    tb <- forkIO $ await prB >>= putMVar v
                    x <- takeMVar v
                    killThread ta
                    killThread tb
                    return $ case x of
                               Left f -> reject f
                               Right p -> resolve p
```

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 `pRace`ing 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 -> hangForever)`, so the final `pRace` function is as follows:

```
pRace :: [Promise f p] -> IO (Promise f p)
pRace [] = newPromise (\s f -> hangForever)
pRace (x:xs) = do
  prs <- pRace xs
  pRace2 x prs
```

[implementation of JS's Promise.any(iterable)] Yet another way to combine any number of promises in parallel by executing each simultaneously is `Promise.any(iterable)`. 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.]

Future Work

[decouple Promise monad from IO ?]