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. Additionally, `Promises` can be combined in parallel, with a variety of distinct semantics such as waiting for the first success or for the first completion irrespective of success or failure.

Madsen et al. (2017) note that programmers often make mistakes when writing code involving promises and there are no static checks to detect them.

We build a model of `Promises` in Haskell and discuss what we gain by using a stronger type system. In particular, we show that some classes of errors can be caught automatically by the type checker.

Motivation

`Promises` as adopted by Javascript use model initially proposed in Friedman & Wise (1978).

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 monadic bind ($\gg=$), 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

We will refer to types in Haskell syntax. We may say that a value `v` has type `T` and write this as `v :: T`. Specific types are capitalized, e.g. `3 :: Integer`, while type variables are written in lowercase and are usually a single character. Some types are paramterized by other types. For instance, Haskell lists are linked lists of values that share a type. A list containing values of type `T` will itself have the type `[T]`, that is, list-of-T. Perhaps the most common variety of parameterized type to see in Haskell type signatures is the function, written with an ASCII arrow (`->`). Functions are parameterized by the types of both their input and output; a function that counts the lengths of `String`s could have the type `String -> Integer`. The syntax for calling functions is simple juxtoposition: the expression `f x` means the function `f` applied to `x`. Functions with more than one argument are *curried*, for example a two argument function that takes an `A` and a `B` as input to produce a `C` is written with the type `A -> B -> C`, i.e., a function that accepts an `A` and returns another function that accepts a `B` and returns a `C`[1]. Generalized Algebraic Data Type, or GADT, syntax allows us to specify our own types and give the name and type for each *constructor* function that can create a value of that type. We will write one defining the type `Promise f p` which will represent a `Promise` that yields a result with the type `p` if it succeeds or one of type `f` on failure.

We will discuss what types various objects should have and will, at times, need a convention to refer to a parameterized type when we haven't yet decided what the parameter will be. In these cases, `?` is a metavariable representing some concrete type yet to be decided, rather than real syntax. `Integer -> ? -> ?` means a function accepting an `Integer` and some type we will decide later that returns another value of that type. This is distinct from `Integer -> a -> a` using

[1] `->` associates to the right to facilitate writing curried functions without parentheses

type variables because the latter is a function accepting an `Integer` and any value whatsoever, returning a value of the same type as the input.

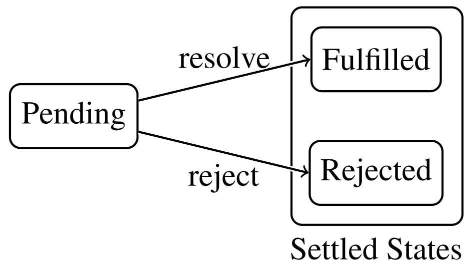
Values in Haskell are referentially transparent, meaning that if an expression evaluates to a value, it can be replaced by that value with no change to the program's semantics. This makes it easier to reason about and prove properties of programs, but comes with a few challenges. It wouldn't do to have a print statement be replaced with its value without executing. Haskell solves this problem with the `IO` monad. A value of type `IO T` can be thought of as an action that may have side effects and will result in a value of type `T`. An action that is performed solely for its side effects and has no useful result value is conventionally given the type `IO ()`. The type `()`, pronounced "unit" is the type of tuples of zero objects. There is only one value of this type, also written `()`. The contents of `IO` values can only be interacted with through the interface of the monad typeclass. In particular, there is no way to convert an `IO T` into a `T`; you cannot get a value out of `IO`. The *do notation* provides a friendly, imperative-looking syntax for interacting with monadic values. In code like the following, the expression `ioThing` returns a value wrapped in a monad.

```
do
  x <- ioThing
  . . .
```

The arrow (`<-`) binds the value so wrapped to the variable `x` and in the region of the code marked by elipses (`. . .`), `x` can be treated as a normal non-monadic value. However, at the end of the block, the final expression must be re-encapsulated into the monad. The simplest way to do this is the function `return`. `return x` simply puts the value `x` into a default context in the monad.

In order to implement `Promises`, we need asynchronous action. To accomplish this in Haskell, we will use `forkIO` and `MVars`. The function `forkIO :: IO () -> IO ThreadId` accepts an `IO ()` action and runs it in a separate thread. In order to communicate between threads, we use `MVars`. A value of type `MVar T` is a thread safe place to store up to one value of type `T`. We interact with `MVars` with three functions: `newEmptyMVar :: IO (MVar a)` creates an `MVar` with no contents. The type variable `a` will usually be inferred by what type we try to add later. `putMVar`

Figure 1: states and transitions



`:: MVar a -> a -> IO ()` accepts a value of type `a` and stores it in the `MVar`. If the `MVar` already contains a value, the thread running `putMVar` will block until the `MVar` is empty. The last, `takeMVar :: MVar a -> IO a` reads an `IO a` from the `MVar`, leaving it empty, and will block, if necessary, until there is a value in the `MVar` to be read before doing so.

Promises

`Promise`s 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.

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 `Promise`s 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 was the cause of multiple errors in the case study from `madsen`. 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. We name this type `Token` because its only value is that you need one to write an `executor` function. 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 Then?

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[2]. 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 we need to be able to read the `state` `MVar`), 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')
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

[2] 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!

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.

```
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.

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 yes, 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. 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` to settle. The return type of `yes` should match that of runPromise, so `yes :: p -> IO ?`. There are no other restrictions on ?, so we canc choose `yes :: p -> IO a`. `no` must also match return types so `no :: 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 argument form of Javascript's `Promise.then()`, adding to the chain in both the success case and the failure case.

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
```

It is sometimes helpful to run a `Promise` to completion to yield a non-`Promise` value storing the results. A function to do so has the type `Promise f p -> IO (Either f p)`. The result must be in `IO` and is either a `Left f` representing failure with the given reason or a `Right p` representing success with the given value. Implementing such a helper function is as simple as calling `runPromise` and passing in a `yes` that wraps its input in `Right` and `IO` and a `no` that wraps in `Left` and `IO`. We call this function `await` in analogy to the *await* keyword in Javascript. Similarities include that it converts from a `Promise` to a non-`Promise` value by waiting for it to complete and that it can only be used inside the appropriate context; either the `IO` monad or an *async* function, as appropriate.

Instances

We noted earlier that `pThen` had a form reminiscent of a monadic bind operation; it is now time to demonstrate the connection more directly by writing a Monad instance for `Promises`. This will, among other things, allow us to use *do notation* when code employing `Promises`. The `Monad` typeclass operates on types of kind `* -> *`, i.e. type "containers" that are parameterized by exactly one other type. But `Promise` takes two type parameters, having kind * -> * -> *. We can fix this mismatch by defining an instance for the partially applied type `Promise f` that has already taken one type parameter. Because `f` is fixed in the instance, a given invocation of a function from one of our instances will need to keep the type of the failure value constant, even if it changes the type of the success value.[3] 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

[3]This is the reason the failure type is specified before the success type in `Promise f p`: it is more straightforward to write instances where the first parameter is held constant and being able to change the success type with `fmap` is more useful.

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 need something 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`. For this case, we can pattern match to `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` [4]. 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`:

[4] for this to fail to terminate, we would need to be trying to run a `Promise` with an infinite number of functions mapped over it

```
  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`.

The instance for `Monad (Promise f)` requires `return :: a -> Promise f a` that puts a value into a neutral context; this can be the same as `pure` from `Applicative`. The other function required to declare a Monad instance is `(>>=)` (pronounced "bind") and, when specialized to `Promise f`, has the type `Promise f a -> (a -> Promise f b) -> Promise f b`. That is, it accepts a `Promise`, `p`, as well as a function, `k`, that converts from a plain value of the type of a sucessful result from `p`. Then, `(>>=)` applies `k` to `p` as though `p` were a plain value instead of a `Promise`. The type of `(>>=)` is exactly that of `pThen` except that it references unadorned `Promises` in the places where `pThen` had `IO Promises`. As will be familiar by now, we must add a new `Promsie` constructor so we can delay execution until `runPromise`. In this case, we can add

```
    PromiseJoin :: Promise f (Promise f a) -> Promise f a
```

which collapses a two-layer `Promise` into a single layer. Join is an equivalent characterization to `(>>=)` as we can write

```
p >>= k = PromiseJoin (fmap k p)
```

using `fmap` to apply `k` to `p` before returning to a single layer of `Promise` with `join`.

To implement `pJoin`, we need to squash a double-decker `Promise`, `pp :: Promise f (Promise f a)`, down to a single layer. We can do this with `pThen`. Normally, `pThen` expects the function argument, `k`, to be a function that adds a layer of `Promise`ness to the input (as well as an IO wrapper): after all, it has the type `p -> IO (Promise f p')`. But what if we instead *only* wrap `k`'s input in `IO`? Here `k x = return x`, with the type `b -> IO b`, but `k` also has to match the type `p -> IO (Promise f p')`, and we know that `pp :: Promise f p`, so `p` $\sim$ `Promise f a`. Because both expressions for `k` have to match for the result to typecheck, `Promise f a -> IO (Promise f p')` $\sim$ `b -> IO b` meaning `a` $\sim$ `p'`. Since the return type is `IO (Promise f p')`, we get out an `IO`-wrapped single-layer `Promise`. In effect we have "tricked" `pThen` into unwrapping a layer of `Promise` by failing to add a layer in a place where it expected us to.

```
pJoin :: Promise f (Promise f p) -> IO (Promise f p)
pJoin pp = pThen pp return
```

Then we can exend `runPromise` like so:

```
runPromise yes no (PromiseJoin pp) = do
p <- pJoin pp
runPromise yes no p
```

While we are adding constructors, let's include one for a dual `Promise`, one that swaps its success and failure types. We can use this if we ever need to `fmap` or »= over the failure value of a `Promise` rather than the success value.

```
PromiseInvert :: Promise p f -> Promise f p
```

The new case to `runPromise` merely swaps the positions of the `yes` and `no` functions so they apply to the correct arguments:

```
runPromise yes no (PromiseInvert pr) = runPromise no yes pr
```

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)`. This function works similarly to the `race` function used to define `amb`, the amiguous choice operator, in Elliott (2009). 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
```

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. The type signature for the binary variation is `pAny2 :: Promise f p -> Promise f' p -> IO (Promise (f, f') p)`. 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. The success types must still be identical as the resulting `Promise` must have a value of that type to succeed and it could come from either input `Promise`. We still need an `MVar`, `v`, 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. We need communication between the forked threads that doesn't interfere with `v`, hence a second `MVar` that the main thread doesn't touch at all. One fork, if it fails, writes to the error `MVar`, while the other waits to read from it after a failure. This ensures that it won't attempt to write a failure value into the result `MVar` unless both forks have failed. The main thread waits to read a value from `v`, then kills both threads since the remaing thread does not need to continue if we have had a success. Since `v` must be able to hold the value of a success or two failures, it needs to have the type `MVar (Either (f, f') p)`.

```
pAny2 :: Promise f p -> Promise f' p -> IO (Promise (f, f') p)
pAny2 prA prB = do v <- newEmptyMVar
                   errA <- newEmptyMVar
                   ta <- forkIO $ runPromise (putMVar v . Right) (putMVar errA) prA
                   tb <- forkIO $ runPromise (putMVar v . Right)
                         (\b -> do a <- takeMVar errA
                                   putMVar v $ Left (a, b)) prB
                   x <- takeMVar v
                   killThread ta
                   killThread tb
                   return $ case x of
                              Left (a, b) -> reject (a, b)
                              Right p -> resolve p
```

The base case of *n*-ary `pAny` is that a an empty list yields a failed `Promise` with an empty list of reasons since we don't have a successful result to show. In the recursive case, we call `pAny` on the tail of the list, then `pAny2` the head of the input list to the result. At this point, we have a `Promise (f, [f]) p`, so we need a way to cons together the pieces of the list of failure reasons in the failure case. Since we only need to touch the result if we have a failure, `pCatch` suffices.

```
pAny :: [Promise f p] -> IO (Promise [f] p)
pAny [] = return $ reject []
pAny (x:xs) = do
  prs <- pAny xs
```

```
  pr <- pAny2 x prs
  pCatch pr (return . reject . uncurry (:))
```

The last of the parallel combiners is `Promise.all(iterable)`. It is a mirror image to `Promise.any`, in that it immediately rejects whenever it encounters the first failure and only succeeds when all of its inputs succeed. `pAll2` and `pAll` are dual to `pAny2` and `pAny`; we can implement them either by duplicating the code and inverting all the tests or by using `PromiseInvert` to switch the true and false cases of the input promises, then switching back after running them through the dual function.

```
pAll2 :: Promise f p -> Promise f p' -> IO (Promise f (p, p'))
pAll2 prA prB = fmap PromiseInvert (pAny2 (PromiseInvert prA) (PromiseInvert prB))

pAll :: [Promise f p] -> IO (Promise f [p])
pAll [] = return $ resolve []
pAll (x:xs) = do
  prs <- pAll xs
  pr <- pAll2 x prs
  pThen pr (return . resolve . uncurry (:))
```

Conclusions and Future Work

Madsen et al. (2017) performed a case study of recent questions posted to the forum StackOverflow about Javascript `Promises`. Out of 21 questions included in the analysis, six were identified as having a root cause of an unintentional return of `undefined`. A type mismtach of this sort, between what is being returned by a function and what is expected elsewhere, is detected at compile time in Haskell, without even needed to run the code and compare actual output to expected output. A further three questions are classified with a bug type of “Dead Promise” meansing that a `Promise` was neither resolved nore rejected, in one case on only some code paths. Our system detects these problems, again statically at compile time, unless the user explicity creates a `Token` value, perhaps by calling `hangForever`.

One useful extension to this work would be to encode more information into the type system in a way that could detect additional classes of errors. The case study from Madsen et al. (2017) includes multiple instances of a programmer attempting to resolve a `Promise` multiple times, which would not be detected by our system at compile time. This class of error could in principle be detected with linear types.

Another potential improvement would be to rearchitect the system so that `Promise` is less strongly coupled to `IO`.

Bibliography

Elliott, C. M. (2009). Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09* (pp.$\tilde{2}$5). Edinburgh, Scotland: ACM Press.

Friedman & Wise (1978). Aspects of Applicative Programming for Parallel Processing. *IEEE Transactions on Computers*, C-27(4), 289–296.

Madsen, M., Lhoták, O., & Tip, F. (2017). A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 1–24.