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 Haskell Implementation

[core bits]

[then and catch]

[runpromise as helper function]

[Functor, Applicative, Monad instances; addition to the type so they don't need to be in IO]

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 to 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 (\f s -> 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.]

4 Comparison with Existing Concurrency Frameworks

[Compare to push-pull frp] [locks]

5 Future Work

[decouple Promise monad from IO ?]