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]

[implementation of JS's Promise.any(iterable)] Javascript's promise API 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 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)`]

4 Comparison with Existing Concurrency Frameworks

[Compare to push-pull frp] [locks]

5 Future Work

[decouple Promise monad from IO ?]