Promise Land

Proving Correctness with Strongly Typed Javascript-Style Promises

Andrei Elliott

May 6, 2022

Promise Land

Introduction

Javascript Promises

Javascript Promises

    model for asynchronous code

Javascript Promises

model for asynchronous code

replaces the "callback Hell" of event-driven programming

Introduction

Javascript Promises

model for asynchronous code

replaces the "callback Hell" of event-driven programming

nicer to use than forks and locks

Javascript Promises

    model for asynchronous code

    replaces the "callback Hell" of event-driven programming

    nicer to use than forks and locks

my contribution

Javascript Promises

   model for asynchronous code
   replaces the "callback Hell" of event-driven programming
   nicer to use than forks and locks

my contribution

   Haskell library for Promises

Javascript Promises

>
> model for asynchronous code
>
> replaces the "callback Hell" of event-driven programming
>
> nicer to use than forks and locks

my contribution

>
> Haskell library for Promises
>
> > can use Promises from Haskell code

Javascript Promises

    model for asynchronous code

    replaces the "callback Hell" of event-driven programming

    nicer to use than forks and locks

my contribution

    Haskell library for Promises

        can use Promises from Haskell code

        correctness checks JS doesn't have

Background

adopted in Javascript in ECMAScript 6 standard (2015)

Promises

adopted in Javascript in ECMAScript 6 standard (2015)

useful, but somewhat error-prone for programmers

Promises

adopted in Javascript in ECMAScript 6 standard (2015)

useful, but somewhat error-prone for programmers

  no static checks

adopted in Javascript in ECMAScript 6 standard (2015)

useful, but somewhat error-prone for programmers

   no static checks

use `then` and `next` to attach handlers to a Promise

Promises

adopted in Javascript in ECMAScript 6 standard (2015)

useful, but somewhat error-prone for programmers

> no static checks

use `then` and `next` to attach handlers to a Promise

> handlers run after a Promise has succeeded or failed

Promises

adopted in Javascript in ECMAScript 6 standard (2015)

useful, but somewhat error-prone for programmers

    no static checks

use `then` and `next` to attach handlers to a Promise

    handlers run after a Promise has succeeded or failed

    result is a new Promise

Promises

adopted in Javascript in ECMAScript 6 standard (2015)

useful, but somewhat error-prone for programmers

    no static checks

use `then` and `next` to attach handlers to a Promise
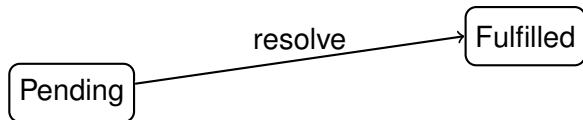
    handlers run after a Promise has succeeded or failed

    result is a new Promise
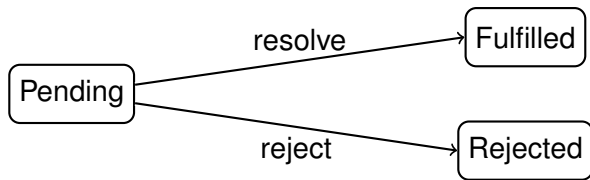
    the computations are said to be *chained* together

Pending

States of a Promise
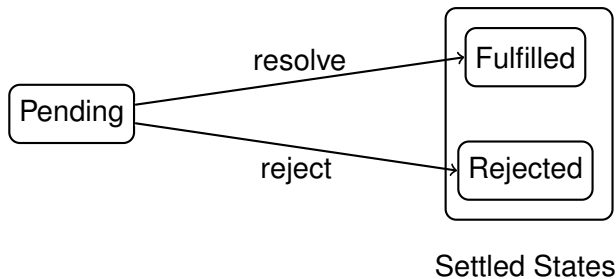

Settled States
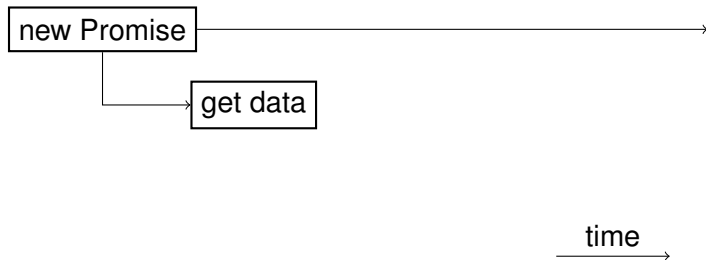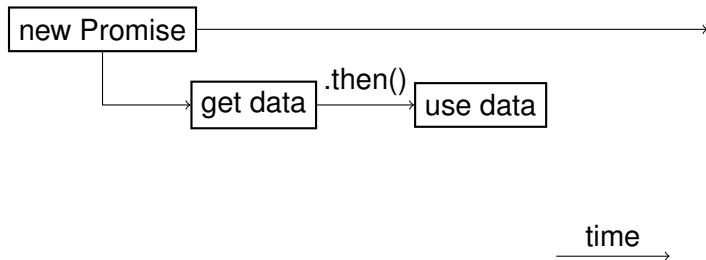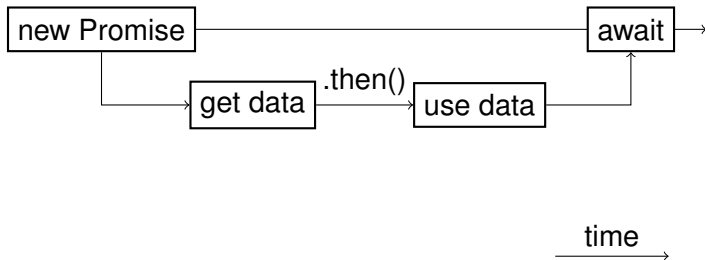
referential transparency

referential transparency

strong type system lets us encode useful information in the types assigned to each value

referential transparency

strong type system lets us encode useful information in the types assigned to each value

  automatically checked by the compiler

referential transparency

strong type system lets us encode useful information in the types assigned to each value

>   automatically checked by the compiler

parameterized types

referential transparency

strong type system lets us encode useful information in the types assigned to each value

    automatically checked by the compiler

parameterized types

    abstract over any type

referential transparency

strong type system lets us encode useful information in the types assigned to each value

    automatically checked by the compiler

parameterized types

    abstract over any type

    ex: `[a]` is a list whose elements have type `a`

referential transparency

strong type system lets us encode useful information in the types assigned to each value

    automatically checked by the compiler

parameterized types

    abstract over any type

    ex: [a] is a list whose elements have type a

    ex: Either a b

        can be a Left a or Right b

Haskell

referential transparency

strong type system lets us encode useful information in the types assigned to each value

    automatically checked by the compiler

parameterized types

    abstract over any type

    ex: `[a]` is a list whose elements have type `a`

    ex: `Either a b`

        can be a `Left a` or `Right b`

        often used as result of a computation that could fail

Monads

Monads

typeclass grouping types with similar behavior

typeclass grouping types with similar behavior

parameterized by one type

typeclass grouping types with similar behavior

parameterized by one type

```
return
    :: a -> m a
```

Monads

typeclass grouping types with similar behavior

parameterized by one type

`return`

> `:: a -> m a`
>
> puts an arbitrary value into a default context

Monads

typeclass grouping types with similar behavior

parameterized by one type

```
return
    :: a -> m a
    puts an arbitrary value into a default context
(>>=)
    :: m a -> (a -> m b) -> m b
```

Monads

typeclass grouping types with similar behavior

parameterized by one type

`return`

> `:: a -> m a`
>
> puts an arbitrary value into a default context

`(>>=)`

> `:: m a -> (a -> m b) -> m b`
>
> combines a monadic value with a function that returns a monad

Monads

typeclass grouping types with similar behavior

parameterized by one type

`return`

> `:: a -> m a`
>
> puts an arbitrary value into a default context

`(»=)`

> `:: m a -> (a -> m b) -> m b`
>
> combines a monadic value with a function that returns a monad

ex: `Either a`

Monads

typeclass grouping types with similar behavior

parameterized by one type

`return`

> `:: a -> m a`
>
> puts an arbitrary value into a default context

`(»=)`

> `:: m a -> (a -> m b) -> m b`
>
> combines a monadic value with a function that returns a monad

ex: `Either a`

> `return = Right`

Monads

typeclass grouping types with similar behavior

parameterized by one type

`return`

    `:: a -> m a`
    puts an arbitrary value into a default context

`(>>=)`

    `:: m a -> (a -> m b) -> m b`
    combines a monadic value with a function that returns a
    monad

ex: `Either a`

    `return = Right`
    `(Right x) >>= f = f x`
    `(Left y)  >>= f = (Left y)`

Haskell is referrentially transparent, so how do we represent side effects?

Haskell is referrentially transparent, so how do we represent side effects?

```
IO a
```

action with possible side effects
results in a value of type a

Haskell is referrentially transparent, so how do we represent side effects?

```
IO a
```

> action with possible side effects
> results in a value of type a
> IO is a monad

Haskell is referrentially transparent, so how do we represent side effects?

```
IO a
```
action with possible side effects
results in a value of type `a`
`IO` is a monad

```
forkIO
```
runs an `IO ()` in a separate thread

Haskell is referrentially transparent, so how do we represent side effects?

IO a

> action with possible side effects
> results in a value of type `a`
> IO is a monad

forkIO

> runs an IO () in a separate thread

MVar a

> Thread-safe storage box for up to one value of type `a`

> Haskell is referrentially transparent, so how do we represent side effects?

    IO a

        action with possible side effects
        results in a value of type a
        IO is a monad

    forkIO

        runs an IO () in a separate thread

    MVar a

        Thread-safe storage box for up to one value of type a
        newEmptyMVar :: IO (MVar a)

> Haskell is referentially transparent, so how do we represent side effects?

IO a

> action with possible side effects
> results in a value of type a
> IO is a monad

forkIO

> runs an IO () in a separate thread

MVar a

> Thread-safe storage box for up to one value of type a
> ```
> newEmptyMVar :: IO (MVar a)
> putMVar :: MVar a -> a -> IO ()
> ```

> Haskell is referentially transparent, so how do we represent side effects?

```
IO a
    action with possible side effects
    results in a value of type a
    IO is a monad
forkIO
    runs an IO () in a separate thread
MVar a
    Thread-safe storage box for up to one value of type a
    newEmptyMVar :: IO (MVar a)
    putMVar :: MVar a -> a -> IO ()
    takeMVar :: MVar a -> IO a
```

Implementation

store an `MVar (Either f p)`

store an `MVar (Either f p)`

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

store an `MVar (Either f p)`

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

JS version accepts an *executor* function

store an `MVar (Either f p)`

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

JS version accepts an *executor* function
    two callbacks: for success and failure

Making Promises

store an `MVar (Either f p)`

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

JS version accepts an *executor* function

two callbacks: for success and failure

use: `newPromise (λ s f -> if error then f("Failed!")`
`else s(value))`

Type:

```
executor ->  (Promise f p)
```

Type:

```
executor -> IO (Promise f p)
```

Type:

```
(successFun -> failFun -> ?)  -> IO (Promise f p)
```

Type:

```
((p -> IO ()) -> (f -> IO ()) -> IO ()) -> IO (Promise f p)
```

newPromise

Type:

```
((p -> IO ()) -> (f -> IO ()) -> IO ()) -> IO (Promise f p)
```

Code:
```
newPromise k = do
    state <- newEmptyMVar
    forkIO $ k (putMVar state .  Right)
               (putMVar state .  Left)
    return (Pending state)
```

Then What?

`then` registers a callback to a succeeding Promise

`then` registers a callback to a succeeding Promise

accepts a Promise and a function that creates a new Promise from a success value

then registers a callback to a succeeding Promise

accepts a Promise and a function that creates a new Promise from a success value

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

Parallel Promises

various ways to combine Promises to run in parallel

Parallel Promises

various ways to combine Promises to run in parallel

ex: `race`

runs a list of input Promises

Parallel Promises

various ways to combine Promises to run in parallel

ex: `race`

> runs a list of input Promises
> stops when the first one settles
> settles with that value

various ways to combine Promises to run in parallel

ex: `race`

    runs a list of input Promises

    stops when the first one settles

    settles with that value

    `pRace :: [Promise f p] -> IO (Promise f p)`

Monad Instance

```
instance Monad (Promise f)
```

`instance Monad (Promise f)`

`return` is the function `resolve`

    creates a Promise that succeeds immediately

Monad Instance

`instance Monad (Promise f)`

`return` is the function `resolve`

    creates a Promise that succeeds immediately

(`»=`) for `Promise f` has type

    `Promise f a -> (a -> Promise f b) -> Promise f b`

Monad Instance

`instance Monad (Promise f)`

`return` is the function `resolve`

    creates a Promise that succeeds immediately

(`»=`) for `Promise f` has type

    `Promise f a -> (a -> Promise f b) -> Promise f b`

    looks a lot like `pThen`

Monad Instance

`instance Monad (Promise f)`

`return` is the function `resolve`

    creates a Promise that succeeds immediately

(`»=`) for `Promise f` has type

    `Promise f a -> (a -> Promise f b) -> Promise f b`

    looks a lot like `pThen`

    but `pThen` results in an `IO (Promise f b)`

`instance Monad (Promise f)`

`return` is the function `resolve`

> creates a Promise that succeeds immediately

(`»=`) for `Promise f` has type

> `Promise f a -> (a -> Promise f b) -> Promise f b`
>
> looks a lot like `pThen`
>
> but `pThen` results in an `IO (Promise f b)`
>
> extra Promise constructor storing the callback
>
>> uses `pThen` when we run the Promise

Results

Would the sytem detect real bugs?

Would the sytem detect real bugs?

Madsen et al. (2017) case study

Would the sytem detect real bugs?

Madsen et al. (2017) case study

    21 Stack Overflow questions about JS Promises

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined`

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

3: dead Promise

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

3: dead Promise

executor never calls either callback

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

3: dead Promise

executor never calls either callback

Promise is *Pending* forever

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

3: dead Promise

executor never calls either callback

Promise is *Pending* forever

catch these by updating the type of `newPromise`

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

3: dead Promise

executor never calls either callback

Promise is *Pending* forever

catch these by updating the type of `newPromise`

```
((p -> IO Token) -> (f -> IO Token) -> IO Token)
    -> IO (Promise f p)
```

Would the sytem detect real bugs?

Madsen et al. (2017) case study

21 Stack Overflow questions about JS Promises

6: unintentional `undefined` ✓

3: dead Promise ✓

executor never calls either callback
Promise is *Pending* forever
catch these by updating the type of `newPromise`

```
((p -> IO Token) -> (f -> IO Token) -> IO Token)
    -> IO (Promise f p)
```

Thank You