Promises vs. Monads

Mark S. Miller
TC39 Meeting May 2013

Some definitions

Not concerned here about thenables.

Promises are nominal and not subclassable

T, U -- fully parametric types

t, u -- parametric non-promise types

Ref<T> is union type of T and Promise<T>

("no one wants")
Unabashed Monadic Promises (UP)

Q.fulfill(T) -> Promise<T> // unconditional lift

Promise<T>.**map**(T -> U) -> Promise<U>

Promise<T>.**flatMap**(T -> Promise<U>) -> Promise<U>

If the flatMap onSuccess argument function returns a non-promise, this would throw an Error, so this type description remains accurate for the cases which succeed.

Q-like Promises (QP)

Q(Ref<t>) -> Promise<t> // autolift

Promise<t>.**then**(t -> Ref<u>) -> Promise<u>
(Ignoring the onFailure callback)

Abashed Monadic Promises (AP)

Q.fulfill(T) -> Promise<T> // unconditional lift

Q(Ref<T>) -> Promise<T> // autolift

Promise<T>.**then**(T -> Ref<U>) -> Promise<U>
(Ignoring the onFailure callback)

Typing imprecise. Fix with deterministic overloading

Abashed Monadic Promises (AP)

Q.fulfill(T) -> Promise<T> // unconditional lift

Q(Promise<T>) -> Promise<T>
Q(t) -> Promise<t>

Promise<T>.**then**(T -> Promise<U>) -> Promise<U>
Promise<T>.**then**(T -> u) -> Promise<u>

Precise either because lowercase means “non promise”.
Or by use of Typescript’s prioritized overloading rule.

AsyncTable in UP style

```
class UAsyncTable<T,U> {
    constructor() {
        this.m = Map<T,U>();
    }
    set(keyP :Promise<T>, val :U) {
        keyP.map(key => { this.m.set(key, val) });
    }
    get(keyP :Promise<T>) :Promise<U> {
        return keyP.map(key => this.m.get(key));
    }
}
```

AsyncTable in QP style

```
class QAsyncTable<t,u> {
    constructor() {
        this.m = Map<t,u>();
    }
    set(keyP :Ref<t>, valP :Ref<u>) {
        Q(keyP).then(key => { this.m.set(key, valP) });
    }
    get(keyP :Ref<t>) :Promise<u> {
        return Q(keyP).then(key => this.m.get(key));
    }
}
```

AsyncTable in AP style, take #1

```
class A1AsyncTable<T,U> {
    constructor() {
        this.m = Map<T,U>();
    }
    set(keyP :Promise<T>, val :U) {
        keyP.then(key => { this.m.set(key, val) });
    }
    get(keyP :Promise<T>) :~~Promise<U>~~ {
        return keyP.then(key => this.m.get(key));
    }
}
```

AsyncTable in AP style, take #2

```
class AAsyncTable<T,U> {
    constructor() {
        this.m = Map<T,U>();
    }
    set(keyP :Promise<T>, val :U) {
        keyP.then(key => { this.m.set(key, val) });
    }
    get(keyP :Promise<T>) :Promise<U> {
        return keyP.then(key => Q.fulfill(this.m.get(key)));
    }
}
```

A conflict of styles

Good QP-style QAyncTable

Considered broken by AP programmer

Good AP-style AAsyncTable

Considered broken by QP programmer

But first…

Loss of QP expressiveness?

The best criticism of QP style is that sometimes you want promises as opaque payloads of promise operations. True. How does the QP programmer cope?

By yet another wrapping/unwrapping strategy.

How QP client copes with QP

```
var qt = QAsyncTable();
… k and payload in scope …
qt.set(k, {value: payload});
…
Q(qt.get(k)).then(wrapper => {
    log(“arrived”);
    return wrapper.value;
});
```

Example from actual code

```
function makeRemote(remoteDispatch, nextSlotP) {
 …
 Q(nextSlotP).then(function(nextSlot) {
       become(handle(remotePromise, Q(nextSlot.value)));
     }, function(reason) {
       become(handle(remotePromise, rejected(reason)));
     }).done();
  …
```

How AP client would cope with QP

```
var qt = QAsyncTable();
… k and payload in scope …
qt.set(k, Q.fulfill(payload));  // delicate
…
Q(qt.get(k)).then(payload => {
    log(“arrived”);
    return payload;
});
```

How AP client would cope with QP

```
var qt = QAsyncTable();
… k and payload in scope …
qt.set(k, {value: payload});  // robust
…
Q(qt.get(k)).then(wrapper => {
    log(“arrived”);
    return wrapper.value;
});
// Same as how QP copes with QP
```

How QP client would cope with AP

```
Var at = AAsyncTable();
… k and payload in scope …
at.set(k, payload);
…
Q(at.get(k)).deepFlatten(payload => {
    log(“arrived”);
    return payload;
});
```

AP2 (Based on Tab's latest)

- Q.fulfill // lifting
- Q() // autolifting, resolve
- p.then // deep flattening
- p.flatMap // "chain"

AP3

- Q.fulfill // lifting
- Q() // autolifting, resolve
- p.then // deep flattening of returned value

Based on Domenic's latest

- M.fulfill = x => Q({value: x});
- M.flatMap = (p,f) =>

```
    p.then(({value:x}) => f(x));
```

Surprising Conclusion

There’s no crisis here after all.

Whichever we do, all can cope.

The coping will be ugly, but reliable patterns are possible.

We’ve survived uglier things.

We already agree on tremendously more than we disagree on!