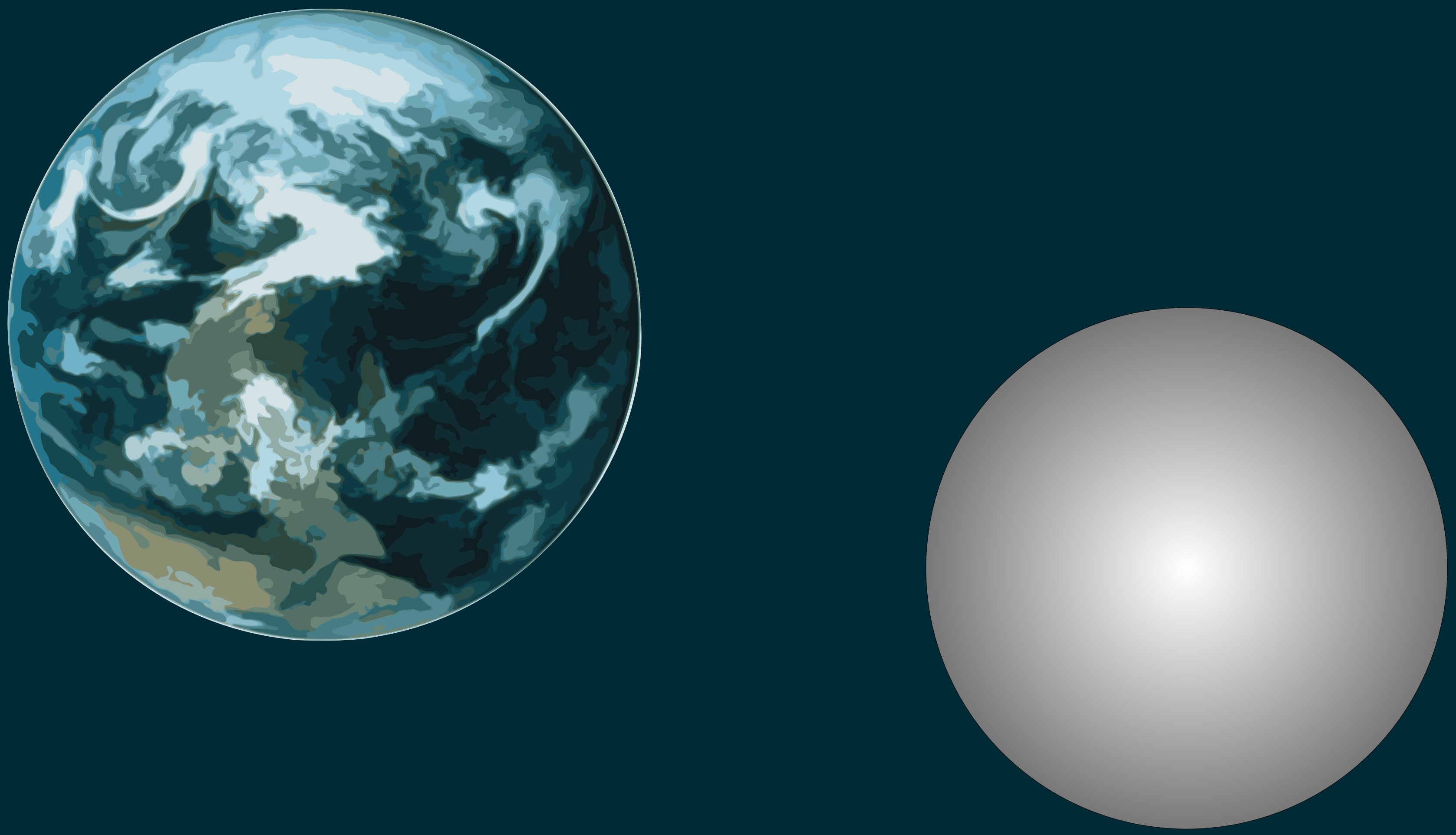
The Mathematical Underpinnings of Promises in C++

David Sankel ⟨dsankel@bloomberg.net⟩
Bloomberg
C++Now 2017

This is a math talk.



Top-down bottom-up

- Math is always an approximation.
- Adjustments necessary for real-world concerns.

This is a promise talk.

JavaScript

```
promise.then(function(result) {
  console.log(result); // Promise was fulfilled
}, function(err) {
  console.log(err); // Promise was rejected
});
```

C++

```
std::promise<int> p;
std::future<int> f = p.get_future();
std::thread( [&p]{ p.set_value(41); }).detach();
```

Promises libraries (generally)

- Include a type representing a value set elsewhere.
- Have a means to "fulfill" the value or signal an error.
- Include functions to build new promises from other promises.
- Replace callbacks.

Denotational Design Review

$$\mu[\![\text{add}(e_1,\ e_2)]\!] = \mu[\![e_1]\!] + \mu[\![e_2]\!]$$

Denotational design

- Top-down.
- Maps syntax to semantics.
- Recursive.

The meaning of a promise/future?

$\mu[\![$ `promise<V>` $]\!] = ?$

A few possibilities

$$\mu[\![\, \texttt{promise<U>} \,]\!] = \mu[\![\, U \,]\!]$$

$$\mu[\![\, \texttt{promise<U>} \,]\!] = \mu[\![\, U \,]\!] + \mathrm{Error}$$

$$\mu[\![\, \texttt{promise<U>} \,]\!] = T \times \mu[\![\, U \,]\!]$$

$$\mu[\![\, \texttt{promise<U>} \,]\!] = T \times (\mu[\![\, U \,]\!] + \mathrm{Error})$$

Operations on promises?

- `fulfill(v)`
- `p.then(f)`
- `all(p₁, p₂)`
- `first(p₁, p₂)`

$$\mu[\![\, \texttt{promise<U>} \,]\!] = \mu[\![\, U \,]\!]$$

Fundamental operations on values

```
pure : U → promise<U>
map : (U → V, promise<U>) → promise<V>
apply : (promise<U → V>, promise<U>) → promise<V>
join : promise<promise<U>> → promise<V>
```

Notation:

- `a : b`. a has type b
- `u → v`. `std::function<v (u)>`
- `(u,v) → w`. `std::function<w (u,v)>`

Identity monad operations

given, μ⟦ `promise<U>` ⟧ = μ⟦ U ⟧

```
μ⟦ pure(v) ⟧        = μ⟦ v ⟧
μ⟦ map(f, p) ⟧      = μ⟦ f ⟧( μ⟦p⟧ )
μ⟦ apply(pf, pu) ⟧ = μ⟦ pf ⟧( μ⟦ pu ⟧ )
μ⟦ join(pp) ⟧       = μ⟦ pp ⟧
```

Implement `pure` with `fulfill`

```
template<typename T>
promise<T> fulfill(T value);
    // Create a new promise that is fulfilled with the specified
    // 'value'.
```

```
template<typename T>
promise<T> pure(V v) {
  return fulfill(v);
}
```

Implement `map` with `then`

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F(T)>> then(F continuation);
      // Return a promise fulfilled with the result of the
      // specified 'continuation' applied to the value of this
      // promise.
}
```

```
template<typename U, typename V>
promise<V> map(std::function<V(U)> f, promise<U> p) {
  // ...
}
```

Implement `map` with `then`

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F(T)>> then(F continuation);
      // Return a promise fulfilled with the result of the
      // specified 'continuation' applied to the value of this
      // promise.
}
```

```
template<typename U, typename V>
promise<V> map(std::function<V(U)> f, promise<U> p) {
    return p.then(f);
}
```

Implement `apply` with `then`

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F(T)>> then(F continuation);
      // Return a promise fulfilled with the result of the
      // specified 'continuation' applied to the value of this
      // promise.
}
```

```
template<typename U, typename V>
promise<V> apply(promise<std::function<V(U)>> f, promise<U> p) {
  // ...
}
```

Implement `apply` with `then` and `get`

```
template<typename T>
class promise {
public:
  T get();
      // Return the value of this promise once fulfilled.
}
```

```
template<typename U, typename V>
promise<V> apply(promise<std::function<V(U)>> f, promise<U> p) {
    return fulfill( f.get()( p.get() ) );
}
```

Second try...

```
template<typename U, typename V>
promise<V> apply(promise<std::function<V(U)>> f, promise<U> p) {
    return f.then([=](std::function<V(U)> f) {
        return f(p.get());
    });
}
```

Can we do better?

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F (T)>> then(F continuation);
      // Return a promise fulfilled with the result of the
      // specified 'continuation' applied to the value of this
      // promise.
}
```

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F (T)>> then(F continuation);
      // Return a promise fulfilled with the result of the
      // specified 'continuation' applied to the value of this
      // promise.

  template<typename F>
  std::result_of_t<F (T)> then(F continuation)
      requires requires (F f, T t) {{f(t) -> promise<auto>}};
      // Return the result of the specified 'continuation'
      // applied to the value of this promise.
}
```

Implement apply

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F (T)>> then(F continuation);

  template<typename F>
  std::result_of_t<F (T)> then(F continuation);
      requires requires (F f, T t) {{f(t) -> promise<auto>}};
}
```

```
template<typename U, typename V>
promise<V> apply(promise<std::function<V(U)>> f, promise<U> p) {
  // ...
}
```

```cpp
template<typename U, typename V>
promise<V> apply(promise<std::function<V(U)>> f, promise<U> p) {
    return f.then( [=](std::function<V(U)> f){
        return p.then( [f](U u) {
            return f(u);
        }
    };
}
```

Oh no, we broke map!

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F (T)>> then(F continuation);

  template<typename F>
  std::result_of_t<F (T)> then(F continuation);
      requires requires (F f, T t) {{f(t) -> promise<auto>}};
}
```

```
template<typename U, typename V>
promise<V> map(std::function<V(U)> f, promise<U> p) {
    return p.then(f);
}
```

```
template<typename T>
class keep {
  typedef T type;
  T t;
};
```

```cpp
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F (T)>> then(F continuation);

  template<typename F>
  std::result_of_t<F (T)> then(F continuation);
      requires requires (F f, T t) {{f(t) -> promise<auto>}};

  template<typename F>
  promise<typename std::result_of_t<F (T)>::type > then(F cont);
      requires requires (F f, T t) {{f(t) -> keep<auto>}};
}
```

```
template<typename U, typename V>
promise<V> map(std::function<V(U)> f, promise<U> p) {
    return p.then([=](U u) {
        return keep{f(u)};
    });
}
```

Now join

```
template<typename T>
class promise {
public:
  template<typename F>
  promise<std::result_of_t<F (T)>> then(F continuation);

  template<typename F>
  std::result_of_t<F (T)> then(F continuation);
      requires requires (F f, T t) {{f(t) -> promise<auto>}};

  template<typename F>
  promise<typename std::result_of_t<F (T)>::type > then(F cont);
      requires requires (F f, T t) {{f(t) -> keep<auto>}};
}
```

```
template<typename T>
promise<T> join(promise<promise<T>> pp) {
    //...
}
```

```cpp
template<typename T>
promise<T> join(promise<promise<T>> pp) {
  return pp.then([](promise<T> p) {
    return p;
  });
}
```

We have a promise interface that forms a monad.

How do we improve the design for C++?

`void` continuation

```
p.then( [](int i) {
  std::cout << "Got an int " << i << std::endl;
});
```

```
promise<std::monostate> r = p.then( [](int i) {
  std::cout << "Got an int " << i << std::endl;
  return std::monostate();
});
```

```
promise<> r = p.then( [](int i) {
  std::cout << "Got an int " << i << std::endl;
});
```

```cpp
promise<> r = p.then( [](int i) {
  std::cout << "Got an int " << i << std::endl;
});

r.then( []() {
  //...
});
```

```
promise<int, std::string> p = /*...*/;
p.then( [](int i, std::string s) {
  // ...
});
```

```
promise<int, std::string> q = p.then( []() {
  return std::make_tuple(3, std::string("hello"));
});
```

then

```
void                         ⇒ promise<>
T                            ⇒ promise<T>
promise<T>                   ⇒ promise<T>
keep<T>                      ⇒ promise<T>
std::tuple<T1, T2, …, Ti>    ⇒ promise<T1, T2, …, Ti>
```

Something's still missing from the semantics

$\mu[\![\texttt{promise<U>}]\!] = \mu[\![\mathrm{U}]\!]$

What about these?

```
all(p₁, p₂)
first(p₁, p₂)
```

`all` looks good

```
all : (Promise<T>, Promise<U>) → Promise<T,U>

μ⟦all(p₁, p₂)⟧ = (μ⟦p₁⟧, μ⟦p₂⟧)
```

first

```
first : (Promise<T>, Promise<U>) → Promise<variant<T,U>>

μ⟦first(p₁, p₂)⟧ = ?
```

Okay, back to the drawing board...

```
μ⟦ promise<U> ⟧ : (T, μ⟦ U ⟧)
```

What is time?

- Seconds since epoch?
- Relative seconds (a real number)?
- Something else?

Time

Lets try $\mathtt{T} \approx \mathbb{N}$.

pure, first, and all

```
pure : U → promise<U>
pure u = (0, u)

first((t₁,v₁), (t₂, v₂)) = if t₁ < t₂ then (t₁,v₁) else (t₂, v₂)
  all((t₁,v₁), (t₂, v₂)) = (max(t₁, t₂), (v₁, v₂))
```

map

```
map : (U → V, promise<U>) → promise<V>
map (f, (t, u)) = (t+1, u)
```

That doesn't work out so well

- We'd have to store `T`
- Order matters for real promises

```
first(map(f, a), map(g, a)) ≈ map(f, a)
first(map(g, a), map(f, a)) ≈ map(g, a)
```

What can we do about sequentiality?

- Pay the time storage cost.
- Treat it as nondeterministic.
- Build an operational semantics for it.

$$\frac{}{\langle E, p_2 = p_1.\mathtt{then}(f); e\rangle \rightarrow \langle E \cup \{p_1 » p_2\}, e\rangle}$$

$$\frac{p_1 » p_2 \in E \quad p_2 » p_3 \in E}{p_1 » p_3 \in E}$$

$$\frac{\langle E, p_1 \rightarrow W\rangle \quad p_1 » p_2 \in E}{\langle E, \mathtt{first}(p_1, p_2)\rangle \rightarrow \langle E, p_1\rangle} \qquad \frac{\langle E, p_1 \rightarrow W\rangle \quad p_2 » p_1 \in E}{\langle E, \mathtt{first}(p_1, p_2)\rangle \rightarrow \langle E, p_2\rangle}$$

$$\frac{\langle E, p_1 \rightarrow v_1\rangle \quad \langle E, p_2 \rightarrow v_2\rangle}{\langle E, \mathtt{first}(p_1, p_2)\rangle \rightarrow \langle E, p_1\rangle}$$

Top-down bottom-up design

- Find the mathematical essence.
- Apply it to a bottom-up design.
- Rinse and repeat.

Benefits

- Means to find minimal set of essential methods.
- Powerful abstractions.

The Mathematical Underpinnings of Promises in C++

David Sankel ⟨dsankel@bloomberg.net⟩
Bloomberg
C++Now 2017