Monads for C++

Bartosz Milewski

Why Monads?

- Programming is composition
- Types make composition safe
  - Most successful program verification technology
- Side effects don’t compose
- Monads: composable side effects
- IO is side effect

IO as Side Effect

- Simplest program:

  1. print “What’s your name?”
  2. name = getLine
  3. print “Hi, “ + name

- No dependency between 1 and 2, yet cannot rearrange
- Inconsistent with STM

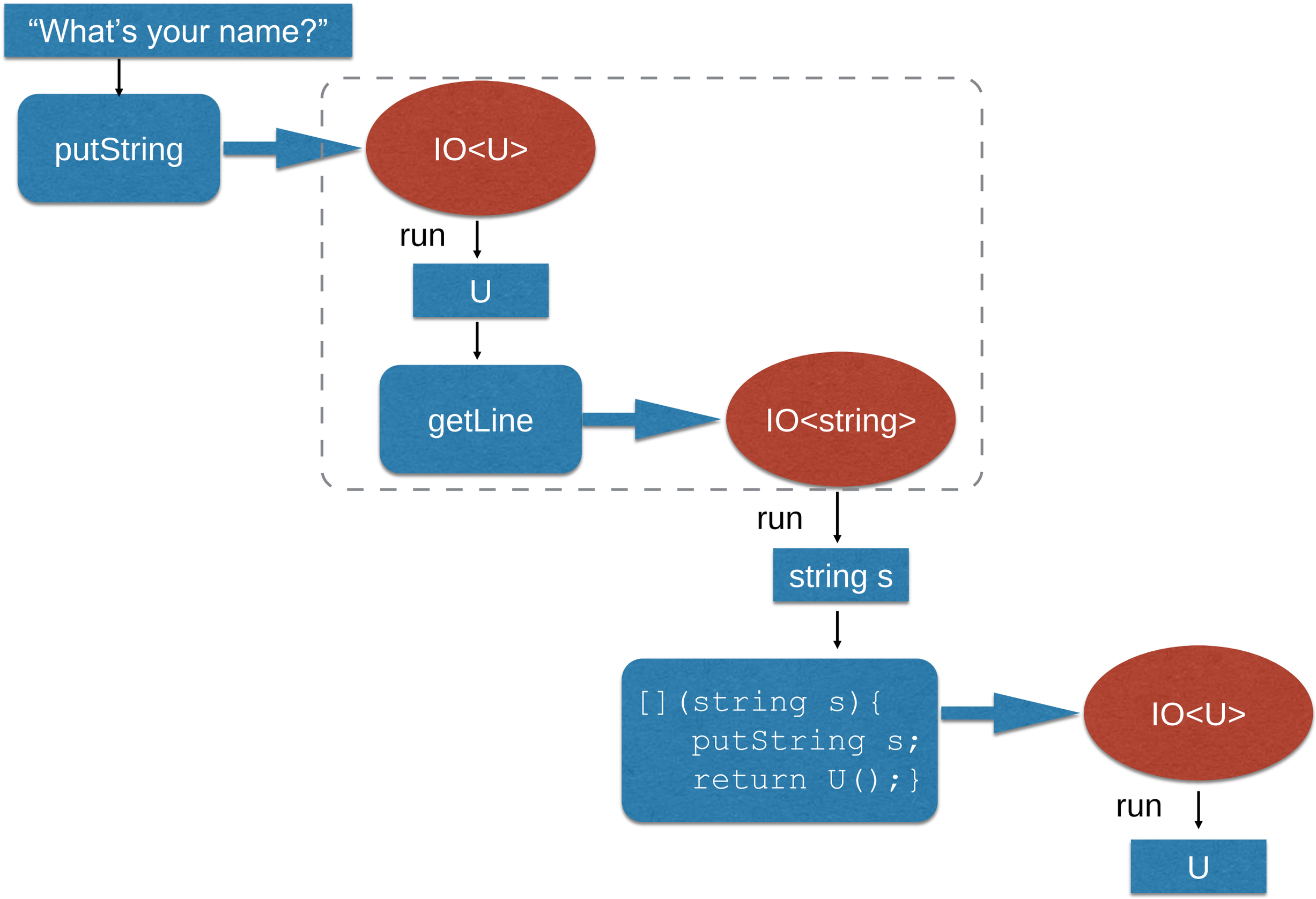
IO monad postpones the action, like a future (or a packaged_task)

```
template<class T>
class IO
{
public:
    IO(std::function<T()> f) : act(f) {}
    T run() { return _act(); }
private:
    std::function<T()> _act;
};
```

```
IO<U> putStr(std::string s)
{
    return IO<U>(
        [s](){ std::cout << s; return U(); }
    );
}
```

```
// Unit type
struct U {};
```

```
IO<std::string> getLine(U)
{
    return IO<std::string>(
        []() {
        std::string s;
        std::getline(std::cin, s);
        return s; }
    );
}
```


“What’s your name?”
putString
IO<U>
run
U
getLine
IO<string>
run
string s
[](string s){
putString s;
return U();}
IO<U>
run
U

```
template<class F>
auto bind(F f) -> decltype(f(_act()))
{
    auto act = _act;
    return IO<decltype(f(_act()).run())>(
        [act, f]() {
            T x = act();
            return f(x).run();
        });
}
IO<U> test()
{
    return putStr("Tell me your name!\n")
        .bind(getLine)
        .bind([](std::string str) {
                return putStr("Hi " + str + "\n"); });
}

void main()
{
   IO<U> io = test();
   io.run();
}
```

```
template<class F>
auto fmap(F f) -> IO<decltype(f(_act()))>
{
    auto act = _act;
    return IO<decltype(f(_act()))>(
        [act, f]() {
           T x = act();
           return f(x);
        });
}
```

```
IO<U> test()
{
    return putStr("Tell me your name!\n")
        .bind(getLine)
        .fmap(upcase)
        .bind([](std::string str) {
                return putStr("Hi " + str + "\n"); });
}
```

```
template<class T>
IO<T> pure(T x) {
    return IO<T>(
        [x]() { return x; });
}

IO<int> guess(int a, int b)
{
    if (a >= b)
        return pure(a);

    int m = (b + 1 + a) / 2;
    return ask(m).bind([=](bool yes) {
                if (yes) return guess(a, m - 1);
                else  return guess(m, b);
            });
}

IO<bool> ask(int i) {
    return putStr("Is it less than ")
        | [i](U) { return putNumber(i); }
        | [](U)  { return putStr(" (y/n)?\n"); }
        | getLine
        | [](std::string s) { return pure(s == "y"); };
}
```

Option

- Computations that may fail
- Normally use exceptions
- Option monad

```
template<class T>
class Option
{
public:
    Option()    : _valid(false) {}
    Option(T t) : _valid(true), _val(t) {}
    template<class F>
    auto bind(F f) -> decltype(f(_val))
    {
        if (_valid) return f(_val);
        else return decltype(f(_val))();
    }
    template<class F>
    auto fmap(F f)->Option<decltype(f(_val))>
    {
        if (_valid)
           return Option<decltype(f(_val))>(f(_val));
        else return Option<decltype(f(_val))>();
    }
private:
    bool _valid;
    T    _val;
};
```

```
template<class T>
Option<T> pure(T t)
{
    return Option<T>(t);
}

Option<double> opSqrt(double x)
{
    if (x >= 0.0) return pure(sqrt(x));
    else return Option<double>();
}

Option<double> opInv(double x)
{
    if (x != 0.0) return pure(1.0 / x);
    else return Option<double>();
}

void test3()
{
    auto y = opSqrt(-1.0)
        .fmap([](double x) { return 2.0 * x; })
        .bind(opInv);
}
```

Vector Monad

- Computations that return many possibilities

- Normally done using nested loops

```
template<class T, class F>
auto bind(std::vector<T> v, F f) -> decltype(f(v[0]))
{
    decltype(f(v[0])) w;
    for (auto i = std::begin(v); i != std::end(v); ++i){
        auto u = f(*i);
        w.insert(end(w), begin(u), end(u));
    }
    return w;
}
```

```
template<class T, class F>
auto fmap(std::vector<T> v, F f)->std::vector<decltype(f(v[0]))>
{
    std::vector<decltype(f(v[0]))> w;
    std::transform(begin(v), end(v), std::back_inserter(w), f);
    return w;
}

template<class T>
std::vector<T> pure(T t)
{
    return std::vector<T>{t};
}
```

```
std::vector<int> triple(int i)
{
    return std::vector<int>{ i - 1, i, i + 1 };
}


int square(int i) { return i * i; }
```

```
void test4()
{
    std::vector<int> v{ 1, 2, 3 };
    auto w = bind(fmap(v, square), triple);
}
```

Monad<T> Pattern

- Monadic functions
  - f(T)->Monad(S)
- Composed using bind:
  - bind (Monad<T>, Monad<S>(T)) -> Monad<S>
- Values modified using fmap
  - fmap(Monad<T>, S(T)) -> Monad(S)
- Default embellishment using pure:
  - pure(T) -> Monad<T>

Don’t Use This Code in Production

- An arbitrary long list of binds will blow your stack
- In an strict language you need arbitrary tail-call optimization (or trampolining)

Thanks to the sponsors!

Bloomberg

JET BRAINS

think-cell

abacogroup.eu

KDAB

RECOGNITION ROBOTICS
The Visual Guidance Company

JFrog | CONAN
C/C++ package manager