Solving A Constraint Satisfaction Problem Using Monads

Bartosz Milewski

Firenze, June 2015

Why Monads

- Higher order control structures
- Monads are to loops like loops are to gotos
- Declarative rather than imperative
- Reusable, composable, factorizable
- Easily parallelizable

Simple Example

```
  s e n d
+ m o r e
----------
m o n e y
```

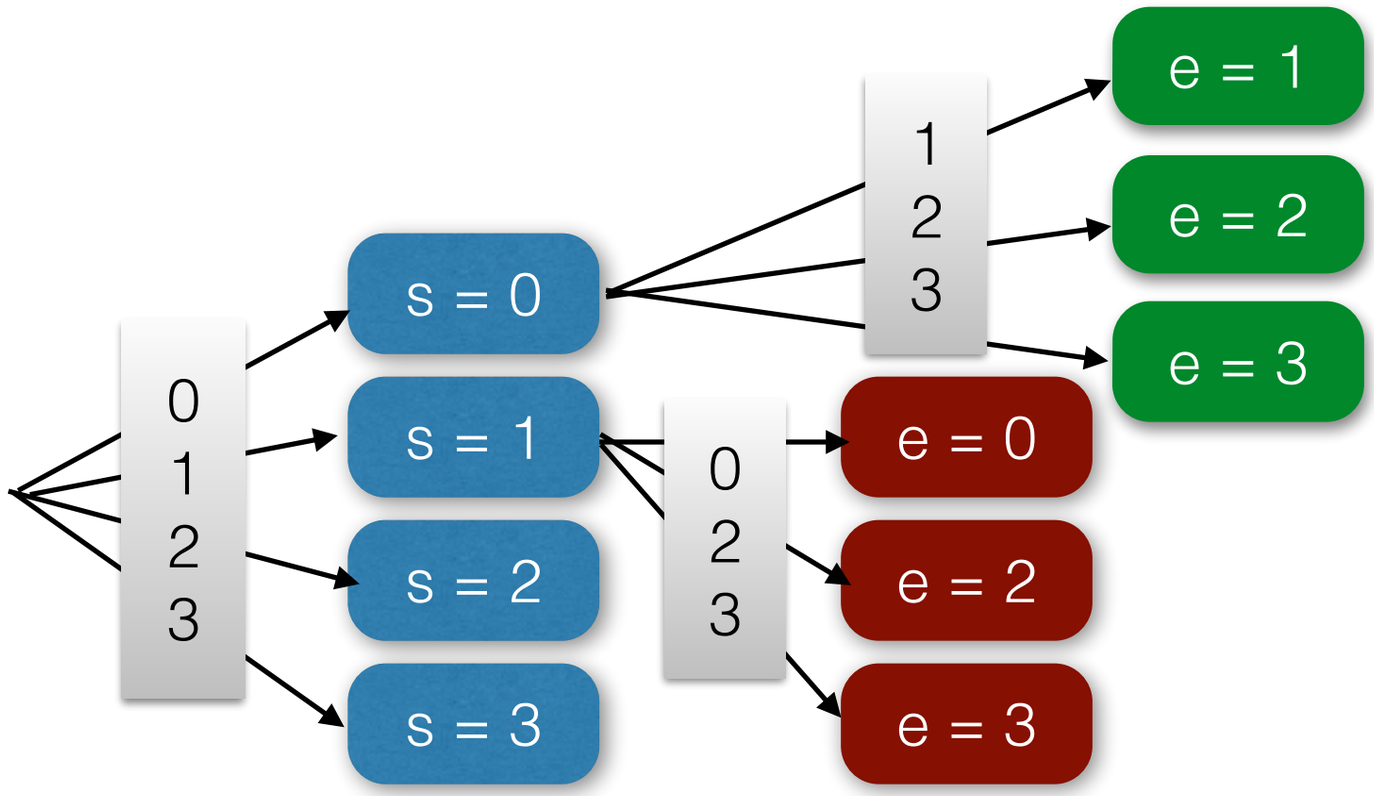
- Each letter different digit
- send + more = money
- s != 0, m != 0

Brute Force Solution

```
for (int s = 0; s < 10; ++s)
    for (int e = 0; e < 10; ++e)
        for (int n = 0; n < 10; ++n)
            for (int d = 0; d < 10; ++d)
                ...
```

```
e != s
n != s && n != e
d != s && d != e && d != n
```

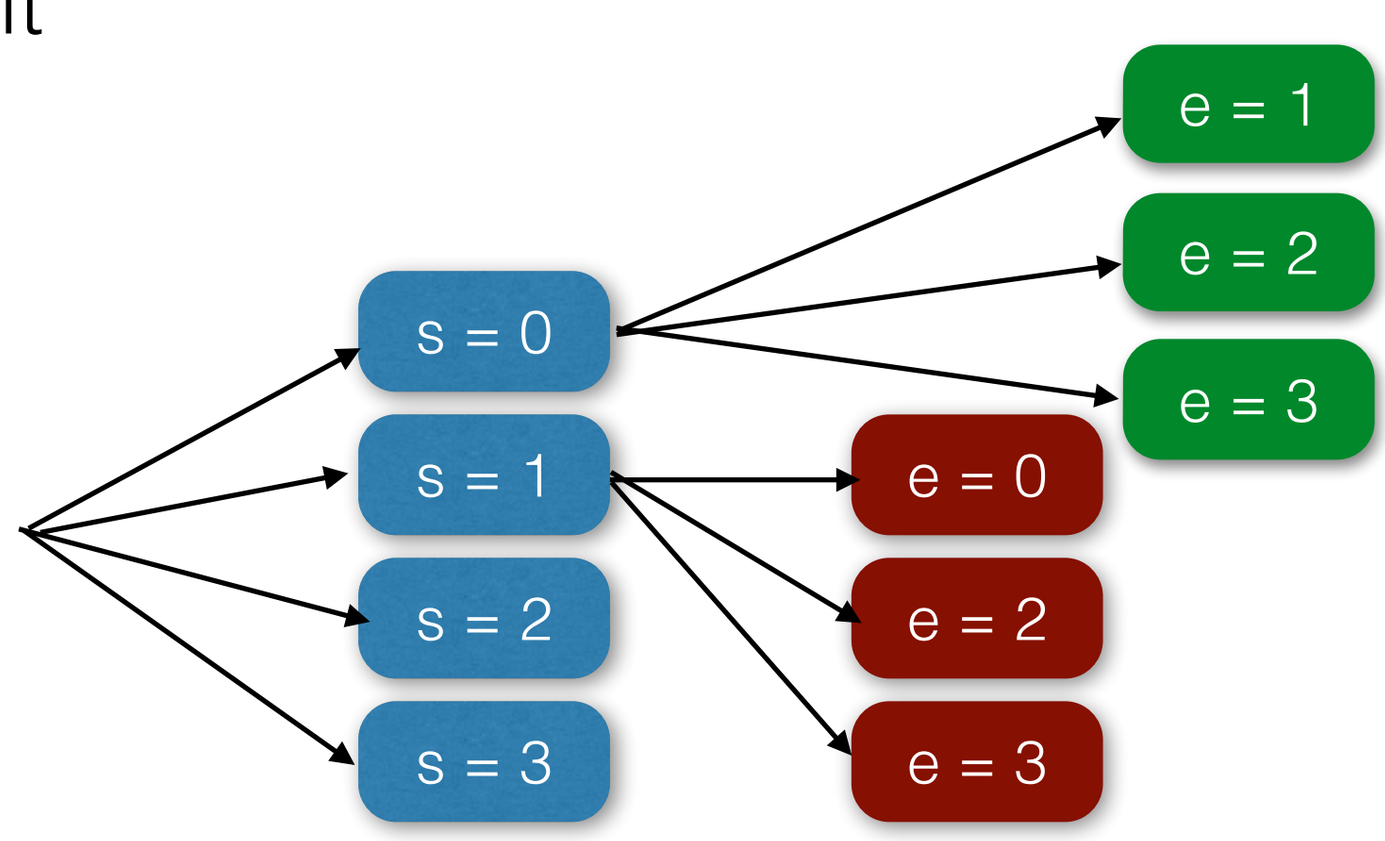
Picking a Number



Quantum Mechanics

Many-World Interpretation

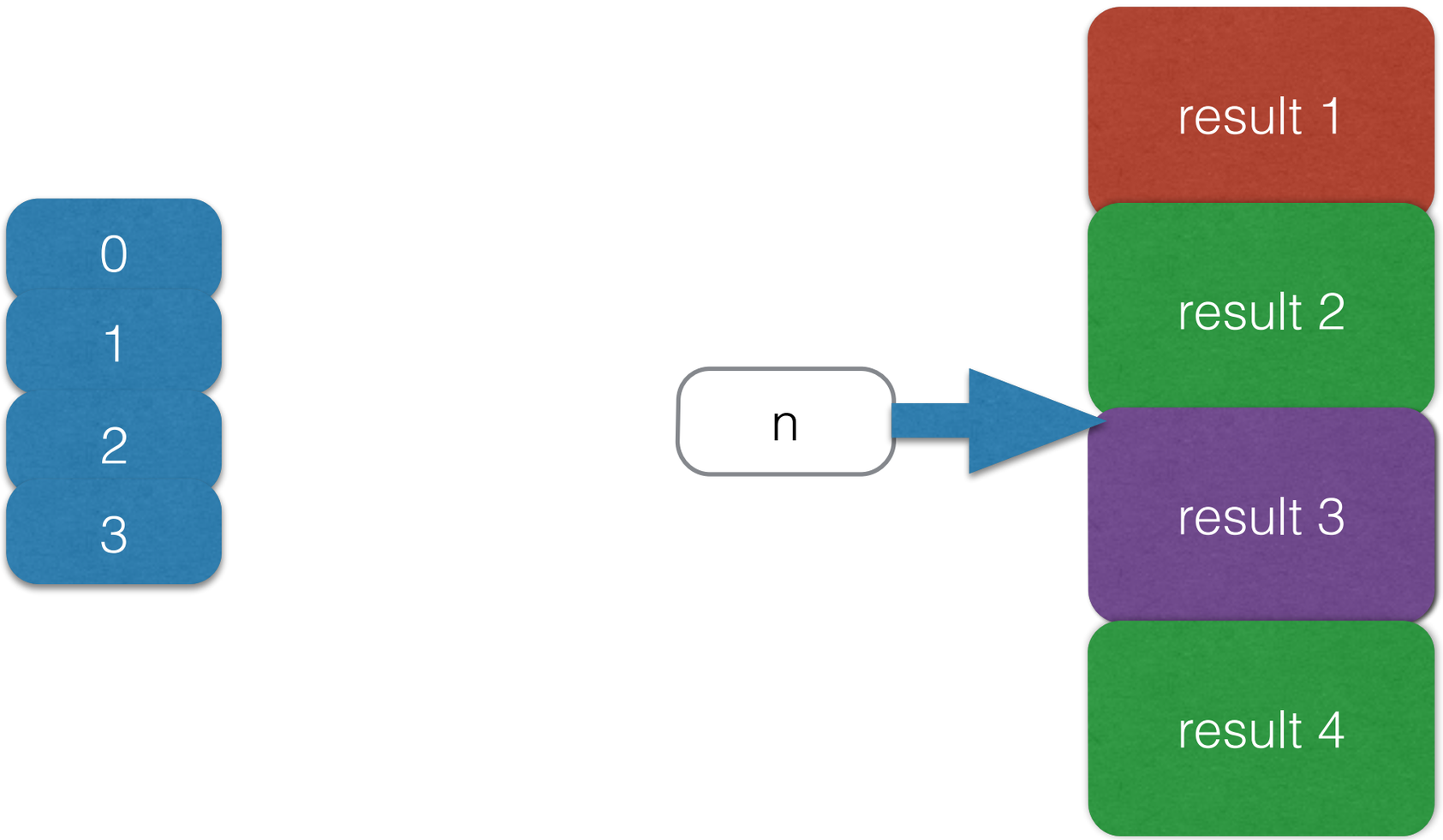
- Each substitution creates a new universe
- Sibling universes share everything except for one digit



Continuation

- Bind the result of previous computation to the rest of the computation

- The continuation also produces multiple universes



Binding

```
template<class A, class B>
List<B> for_each(List<A> lst, function<List<B>(A)> k)
{
    List<List<B>> lstLst = fmap(k, lst);
    return concatAll(lstLst);
}
```

- **`List<A>`** is a persistent list of **`A`**
- **`fmap`** is like a non-destructive **`std::transform`**
- **`concatAll`** concatenates a list of lists

Binding Elaborated

```
template<class A, class F>
auto for_each(List<A> lst, F k) -> decltype(k(lst.front()))
{
    using B = decltype(k(lst.front()).front());
    // This should really be expressed using concepts
    static_assert(std::is_convertible<
        F, std::function<List<B>(A)>>::value,
        "for_each requires a function type List<B>(A)");

    List<List<B>> lstLst = fmap(k, lst);
    return concatAll(lstLst);
}
```

Yielding

```
template<class A>
List<A> yield(A a)
{
    return List<A> (a);
}
```

- Return a singleton list

List Monad

```
template<class A, class B>
List<B> for_each(List<A> lst, function<List<B>(A)> k)
{
    List<List<B>> lstLst = fmap(k, lst);
    return concatAll(lstLst);
}
```

```
template<class A>
List<A> yield(A a)
{
    return List<A> (a);
}
```

- A way to compose computations
- A trivial computation

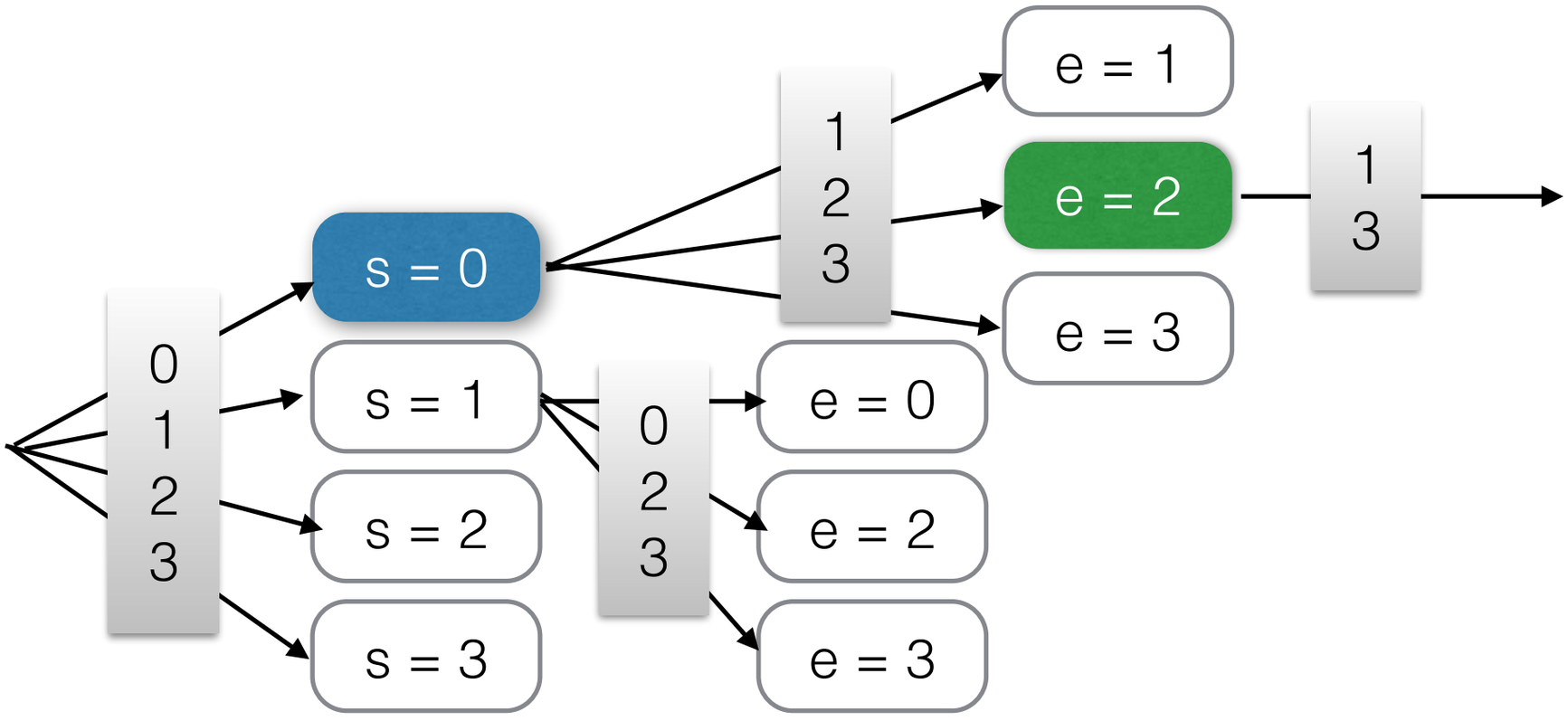
How it's used

```
StateL<tuple<int, int, int>> solve()
{
    StateL<int> sel = &select<int>;

    return for_each(sel, [=](int s) {
    return for_each(sel, [=](int e) {
    return for_each(sel, [=](int n) {
    return for_each(sel, [=](int d) {
    return for_each(sel, [=](int m) {
    return for_each(sel, [=](int o) {
    return for_each(sel, [=](int r) {
    return for_each(sel, [=](int y) {
        return yield_if(s != 0 && m != 0, [=]() {
            int send  = asNumber(vector{s, e, n, d});
            int more  = asNumber(vector{m, o, r, e});
            int money = asNumber(vector{m, o, n, e, y});
            return yield_if(send + more == money, [=]() {
                return yield(make_tuple(send, more, money));
            });
        });
    });});});});});});});});
}
```

Making Plans

List as State




```
pair<A, List<int>> -> pair<B, List<int>>

A -> (List<int> -> pair<B, List<int>>)
```

I have a plan

- If I had lots of cash, I would buy a car, and still have some cash left

```
pair<Car, Cash> buyCar(Cash cashIn)
```

- Special case of a financial plan

```
template<class A>
using Plan = function<pair<A, Cash>(Cash)>;
```

Planning a trip

```
Plan<Trip>(Car)
```

- I can create a financial plan to travel across the US on a budget, provided I have a car

- I have a plan to buy a car

```
pair<Car, Cash> buyCar(Cash cashIn);

Plan<Car> carPlan = &buyCar;
```

- Composition: As soon as I get cash, execute **`carPlan`**, and given the car create a trip plan. Use the leftover cash to execute the trip

Planning with lists

```
using State = List<int>;
```

- A plan takes a list and returns a value paired with a new list

```
template<class A>
using Plan = function<pair<A, State>(State)>;
```

```
template<class A>
pair<A, State> runPlan(Plan<A> pl, State s)
{
    return pl(s);
}
```

Binding

- A plan that produces A
- A continuation that uses A to create a plan to produce B

```
template<class A, class B>
Plan<B> mbind(Plan<A> pl, function<Plan<B>(A)> k)
{
   return [pl, k](State s) {
         pair<A, State> ps = runPlan(pl, s);
         Plan<B> plB = k(ps.first); // give it an A
         return runPlan(plB, ps.second); // new state!
   };
}
```

- Plans don’t run during binding!

A got-it! plan

- A trivial plan to produce something you already have without changing the state

```
template<class A>
Plan<A> mreturn(A a)
{
    return [a](State s) { return make_pair(a, s); };
}
```

The State Monad

```
template<class A, class B>
auto mbind(Plan<A> pl, function<Plan<B>(A)> k)
{
    return [pl, k](State s) {
          pair<A, State> ps = runPlan(pl, s);
          Plan<B> plB = k(ps.first); // give it an A
          return runPlan(plB, ps.second); // new state!
    };
}
```

```
template<class A>
Plan<A> mreturn(A a)
{
    return [a](State s) { return make_pair(a, s); };
}
```

- A way to compose computations
- A trivial computation

Combining Monads

StateList

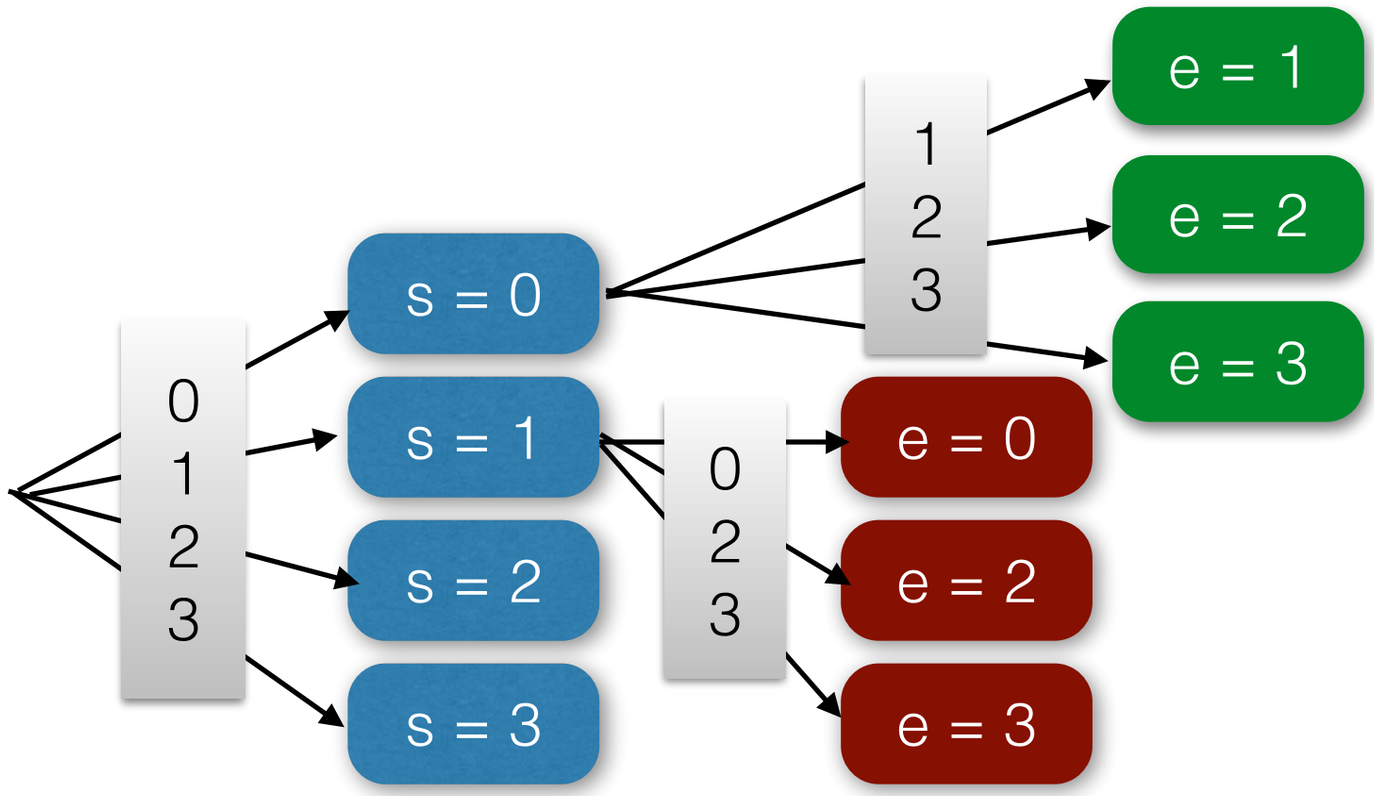
- List of available digits as state
- Functions (plans) that return multiple alternative results, each paired with a new state

```
using State = List<int>;
```

```
template<class A>
using PairList = List<pair<A, State>>;
```

```
template<class A>
using StateList = function<PairList<A>(State)>;
```

Picking a Number



StateList as Monad Plus

- **`mbind`** (to compose a list of plans with a continuation that produces a list of plans)
- **`mthen`** (for continuations that take void argument)
- **`mreturn`** (to create a trivial singleton plan)
- **`guard`** (to either call or abort continuation)

How it's used

```
StateL<tuple<int, int, int>> solve()
{
    StateL<int> sel = &select<int>;

    return mbind(sel, [=](int s) {
    return mbind(sel, [=](int e) {
    return mbind(sel, [=](int n) {
    return mbind(sel, [=](int d) {
    return mbind(sel, [=](int m) {
    return mbind(sel, [=](int o) {
    return mbind(sel, [=](int r) {
    return mbind(sel, [=](int y) {
        return mthen(guard(s != 0 && m != 0), [=]() {
            int send  = asNumber(vector<int>{s, e, n, d});
            int more  = asNumber(vector<int>{m, o, r, e});
            int money = asNumber(vector<int>{m, o, n, e, y});
            return mthen(guard(send + more == money), [=]() {
                return mreturn(make_tuple(send, more, money));
            });
        });
    });});});});});});});});
}
```

Conclusion

- This is just one simple example
- Functional code in C++
  - Reusable, Factorizable, Parallelizable
- Need a library of monads in C++
- Need support for monadic syntax (resumable functions)

Resources

- 4-part blog at http://BartoszMilewski.com
- Code available at https://github.com/BartoszMilewski/MoreMoney
- Persistent data structures: https://github.com/BartoszMilewski/Okasaki