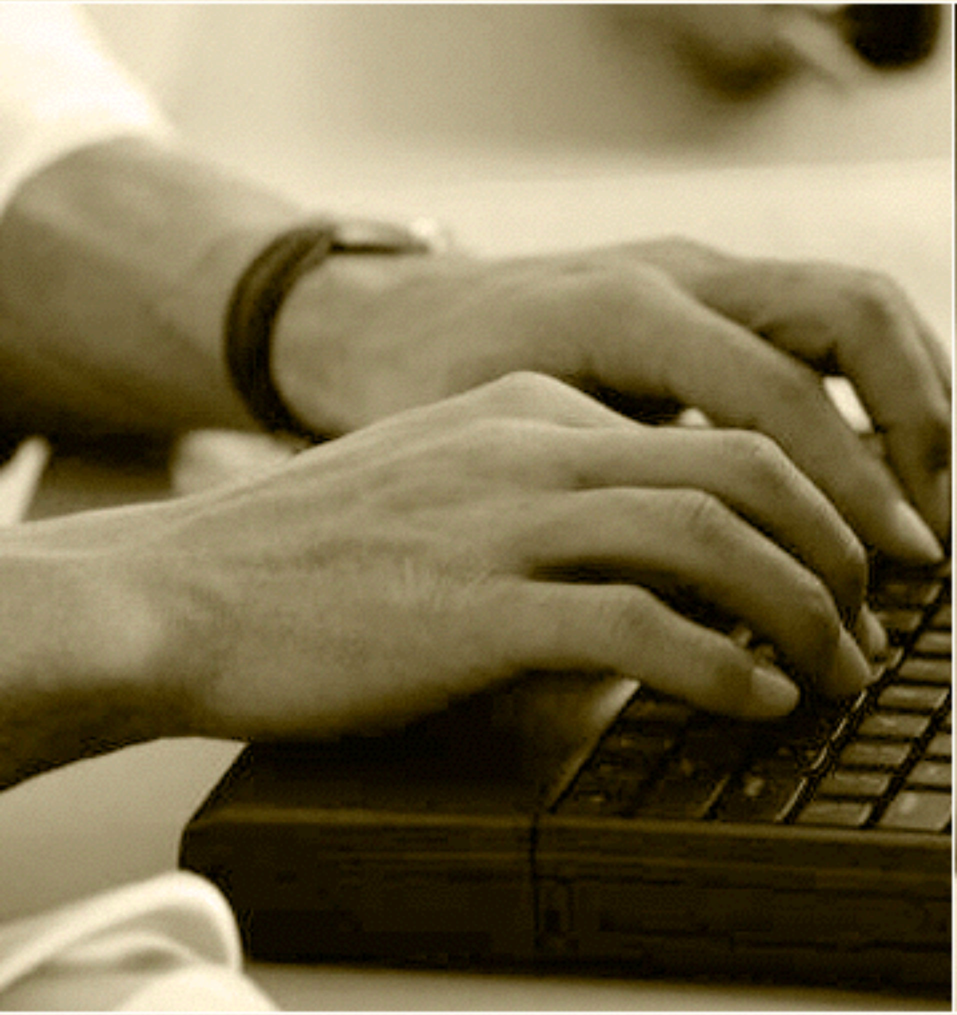
Functional Data Structures

Bartosz Milewski



Paradigms: All about composability

- What's right about OOP?
- What's wrong with OOP?
  - Objects don't compose with concurrency
  - Recipe for data race:
    - Data hiding + Mutation + Sharing
  - Recipe for deadlock:
    - Mutex hiding
    - Locks don't compose

Immutability to the rescue

- Composes with data hiding
- Composes with data sharing
- Requires no synchronization
- Introduces no long-distance coupling
- Functional paradigm allows *controlled mutation*

Persistent data structures

- Replace mutation with construction
- Composition of immutable objects
  - Reuse parts in construction
  - Sharing rather than copying
- Old versions *persist*


Thread safety

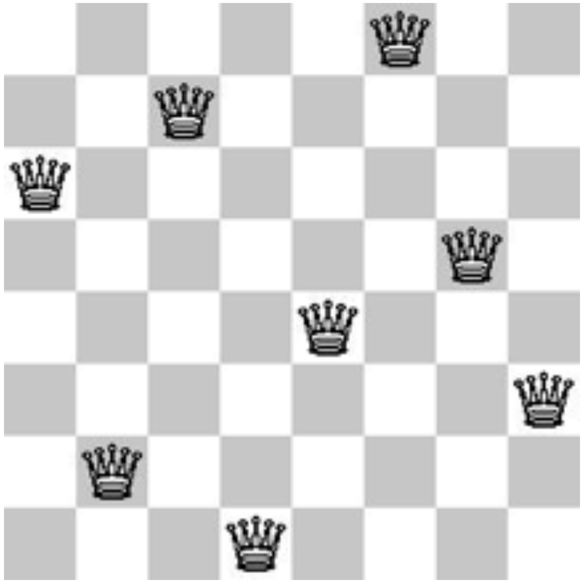
- No data race without mutation
- No data is born immutable (publication safety)
- Resource management
  - Shared pointers
  - Safe lock-free data structures
  - Use make_shared


Example

DOCUMENT


Persistent Document Object

- Document as a persistent tree
- Every edit creates a new version
- Trivial undo, copy/paste between versions
- Concurrent operations in background
  - Spell checking
  - Saving



Persistent data structures use case

EIGHT QUEENS PROBLEM

Divide and Conquer

- Partial solution: k rows with queens
- If k == dim, we are done
- Refinement: Generate partial solutions with an unchecked queen in k+1st row
- Recurse
- Generic divide and conquer pattern

```
template<class Partial, class Constraint>
std::vector<typename Partial::SolutionT> generate( Partial const & part
                                                 , Constraint constr)
{
    using SolutionVec = std::vector<typename Partial::SolutionT>;

    if (part.isFinished(constr))     {
        SolutionVec result{ part.getSolution() };
        return result;
    }
    else {
        List<Partial> partList = part.refine(constr);

        SolutionVec result;
        forEach(std::move(partList), [&](Partial const & part){
            SolutionVec lst = generate(part, constr);
            result.reserve(result.size() + lst.size());
            std::copy(lst.begin(), lst.end(), std::back_inserter(result));
        });
        return result;
    }
}
```


```
class PartSol
{
public:
    typedef List<Pos> SolutionT;

    PartSol() : _curRow(0) {}
    PartSol(int row, List<Pos> const & qs)
        : _curRow(row), _queens(qs) {}

    List<Pos> getSolution() const { return _queens; }
    bool isFinished(int dim) const { return _curRow == dim; }
    List<PartSol> refine(int dim) const;
private:
    bool isAllowed(Pos const & pos) const;

    int       _curRow;
    List<Pos> _queens;
};
```

```
List<PartSol> PartSol::refine(int dim) const
{
    List<PartSol> parts;
    for (int col = 0; col < dim; ++col)
    {
        if (isAllowed(Pos(col, _curRow)))
          parts = parts.push_front(PartSol(_curRow + 1
                                           ,_queens.push_front(Pos(col, _curRow))));
    }
    return parts;
}

List List::push_front(T v) const;
```

- Persistence vs. backtracking

```
template<class Partial, class Constraint>
std::vector<typename Partial::SolutionT> generatePar(int depth, Partial const & part
                                                     ,Constraint constr)
{
    if (depth == 0)
        return generate(part, constr);
    else if (part.isFinished(constr))
        return { part.getSolution() };
    else {
        List<Partial> partList = part.refine(constr);
        std::vector<std::future<SolutionVec>> futResult;
        forEach(std::move(partList), [&, depth](Partial const & part)
        {
            std::future<SolutionVec> futVec =
                std::async([constr, part, depth]() {
                return generatePar(depth - 1, part, constr);
            });
            futResult.push_back(std::move(futVec));
        });
        std::vector<SolutionVec> all = when_all_vec(futResult);
        return concatAll(all);
    }
}
```

The lowest of data structures revisited

PERSISTENT LIST

A list is:

- Empty, or
- An element (head) and a list (tail)

```
template<class T>
class List // as if we had garbage collection
{
    struct Item {
        Item(T v, Item const * tail) : _val(v), _next(tail) {}
        T _val;
        Item const * _next;
    };
    Item const * _head; // null pointer encodes empty list
public:
    List() : _head(nullptr) {}
    List(T v, List tail) : _head(new Item(v, tail._head)) {}
};
```

```
bool isEmpty() const { return !_head; }

T front() const
{
    assert(!isEmpty());
    return _head->_val;
}

List pop_front() const
{
    assert(!isEmpty());
    return List(_head->_next); // private constructor
}

List push_front(T v) const
{
    return List(v, *this);
}
```

Memory Management

```
template<class T>
class List
{
    struct Item
    {
        Item(T v, std::shared_ptr<const Item> const & tail)
            : _val(v), _next(tail) // <- reference count increased
        {}
        T _val;
        std::shared_ptr<const Item> _next;
    };
    std::shared_ptr<const Item> _head;
public:
    List() {}
    List(T v, List const & tail)
        : _head(std::make_shared<Item>(v, tail._head)) {}
};
```


Greed vs. Sloth, Bredth vs. Depth

LAZINESS

```
template<class Partial, class Constraint>
std::vector<typename Partial::SolutionT> generate( Partial const & part
                                                 , Constraint constr)
{
    if (part.isFinished(constr))
        return { part.getSolution() };
    else
    {
        Stream<Partial> partList = part.refine(constr);

        SolutionVec result;
        forEach(std::move(partList), [&](Partial const & part){
            SolutionVec lst = generate(part, constr);
            result.reserve(result.size() + lst.size());
            std::copy(lst.begin(), lst.end(), std::back_inserter(result));
        });
        return result;
    }
}
```

```
Stream<PartSol> PartSol::refineRow(int col, int dim) const
{
    while (col < dim && !isAllowed(Pos(col, _curRow)))
        ++col;
    if (col == dim)
        return Stream<PartSol>();
    return Stream<PartSol>([this, col, dim]() -> Cell<PartSol>
    {
        PartSol part(_curRow + 1, _queens.push_front(Pos(col, _curRow)));
        Stream<PartSol> tail = refineRow(col + 1, dim);
        return Cell<PartSol>(part, tail);
    });
}

Stream<PartSol> PartSol::refine(int dim) const
{
    return refineRow(0, dim);
}
```

Generalization of lazy input range

LAZY STREAM

Lazy stream is:

- Empty, or
- Suspended Cell

```
template<class T>
class Stream
{
private:
    std::shared_ptr <Susp<Cell<T>>> _lazyCell;
public:
    Stream() {}
    Stream(std::function<Cell<T>()> f)
          : _lazyCell(std::make_shared<Susp<Cell<T>>>(f))
    {}
};
```

A Cell is:

- A value and a Stream

```
class Cell
{
public:
    Cell(T v, Stream<T> const & tail)
        : _v(v), _tail(tail) {}
    T val() const { return _v; }
    Stream<T> pop_front() const {
        return _tail;
    }
private:
    T _v;
    Stream<T> _tail;
};
```

Suspended value

- Memoized function

```
template<class T>
class Susp
{
public:
    explicit Susp(std::function<T()> f) : _f(f) {}
    T const & get() {
        std::call_once(_flag, &Susp::set, this);
        return _memo;
    }
private:
    void set() { _memo = _f(); }
    std::once_flag     _flag;
    mutable T          _memo;
    std::function<T()> _f;
};
```

Consuming forEach

```
template<class T, class F>
void forEach(Stream<T> strm, F f)
{
    while (!strm.isEmpty())
    {
        f(strm.get());
        strm = strm.pop_front();
    }
}

Stream(Stream && stm)
   : _lazyCell(std::move(stm._lazyCell))
{}

Stream & operator=(Stream && stm) {
    _lazyCell = std::move(stm._lazyCell);
    return *this;
}
```

Parallel performance

- Conference timetable problem
  - Simon Marlow, Parallel and Concurrent Programming in Haskell
  - Identical divide and conquer skeleton
  - Using persistent red-black tree
  - Parallel and lazy versions
  - Performance

Performance

- Memory management
  - parallel GC vs. reference counting
- Laziness, thunk synchronization
  - call_once vs. lock free, pure function
- Concurrency
  - threads vs. lightweight tasks

Advantages

- Ease of use
  - Implementation follows algorithm
  - Efficient implementation of brute force
- Composability
- Orthogonality
  - Sequential/Parallel
  - Eager/Lazy

Libraries

- https://github.com/BartoszMilewski
- List
- Queue
- Stream
- Red Black Tree (Set and Map)
- Leftist Heap


I see a monad!

EXTRAS

Functor

- Parameterized type
  - Encapsulating a value (values?)

```
template<class T>
class Susp
{
public:
    explicit Susp(std::function<T()> f) : _f(f) {}
    T const & get();
};
```


Functor

- Encapsulating a value (values?)
  - That can be modified by a function

```
template<class T, class F>
auto fmap(Susp<T> susp, F f) -> Susp<(decltype(f(susp.get()))>
{
   using S = decltype(f(susp.get()));
   return Susp<S>([=]() {
        return f(susp.get());
   });
}
```

Monadic functions

- Functions returning monadic values
- User defined functions

```
Susp<vector<int>> ints(int from, int to)
{
   return Susp<vector<int>>([=]() {
       vector<int> v;
       for (int i = from; i <= to; ++i)
          v.push_back(i);
       return v;
   });
};
Susp<int> sum(vector<int> v){
    return Susp([=]() {
       return accumulate(v.begin(), v.end(), 0);
    });
}
```

Composition of monadic functions

```
Susp<vector<int>> ints(int from, int to);
Susp<int> sum(vector<int> v);

Susp<Susp<int>> ssint = fmap(ints(1, 10), sum);
```

- Further composition requires flattening

```
template<class T>
Susp<T> mjoin(Susp<Susp<T>> susp)
{
    return Susp<T>([=]() {
        return susp.get().get();
    }
}

Susp<int> sqsum = fmap(mjoin(fmap(ints(1, 10), sum)), square);
```


Unit of composition

```
template<class T> Susp<T> munit(T v)
{
    return Susp<T>([=]()
    {
        return v;
    };
}
```

```
Susp<string> blah(int i, string s)
{
    if (i == 0) return munit(s + “...”);
    return blah(i - 1, s + “ yada”);
}
```

Monad is Pure Composition

- The essence of reusable composable code
- It can simulate any flow of control
- Its applications are everywhere
  - ranges
  - futures, asynchrony
  - suspensions
  - continuation
  - template expressions