IDENTIFYING MONOIDS

EXPLOITING COMPOSITIONAL STRUCTURE IN CODE

$$\{\mathbb{Z}, \times, 1\}$$

BEN DEANE / @ben_deane / C++NOW / ASPEN, CO / MAY 2019

INTRODUCTION/MOTIVATION

(In response to post-talk questions about how to "identify your monoids")

> *"As a writer of a library, or code that someone else will use, identifying monoids in your code -- in your types and your operations -- I think is one of the single biggest things you can do to help users of your library."*

-- me, Easy to Use, Hard to Misuse: Declarative Style in C++

PRELIMINARIES

Let's get this out of the way.

A monoid is NOT the same thing as a monad.

WHAT IS A MONOID?

> *"Monoidi sunt omnes divisi in partes tres."*

-- Julius Caesar, De Bello Monoido

1. A set of values.
    - finite or infinite
2. A binary operation.
    - closed
    - associative
3. One special value in the set.
    - the identity

A QUICK CODE REVIEW

```
void RepositionListItem(int drag_start_idx, int drop_idx) {
  // grab the dragged item
  Item* item = list_items_[drag_start_index];

  // move the rest down
  for (auto i = drag_start_idx + 1; i < list_items_.size(); ++i) {
    list_items_[i-1] = list_items_[i];
  }
  list_items_.pop_back();

  // re-insert the item ("drop" it)
  list_items_.insert(item, list_items_.begin() + drop_idx);
}
```

- "This is (obviously?) a rotate."
- And there's a bug here. Is `drag_start_idx > drop_idx`?

HOW DID WE KNOW THAT?

Expertise.

- Hard to say how
- Difference in perception
  - Selectivity of attention
  - Units of perception
  - Unconscious search strategies
- Speed of processing

HOW BRAINS WORK

1. See lots of examples with lots of variation. Some variation is relevant, some is not.
2. Figure out which variation is relevant, and classify.
3. Trial and error: rinse, repeat...
4. Result after time: "instinctual" expertise

Katrina Owen *Cultivating Instinct*

https://www.youtube.com/watch?v=Q1Tlo4VnQrA

EXAMPLES

We'll start with the obvious ones

THE OBVIOUS MONOIDS

There's a reason why the default operation of `accumulate` is addition.

- $\{\mathbb{R}, +, 0\}$
- $\{\mathbb{R}, \times, 1\}$

For $\mathbb{R}$, read also $\mathbb{Z}$ or $\mathbb{N}$. (And also $\mathbb{C}$).

ADDITION & MULTIPLICATION

Cover many things that are "number-like".

- integers (approximated by `int` etc)
- real numbers (approximated by `float` or `double`)
- complex numbers
- vectors (in the mathematical sense)
- matrices

We can use (almost) any of these with `accumulate` (or fold expressions) and `plus` or `multiplies`.

`min` AND `max`

It's clear that `max` is a monoid on positive numbers:

$$\{\mathbb{Z}^+, max, 0\}$$

`min` is less clear mathematically...

$$\{\mathbb{Z}, min, ?\}$$

... but we can often use `numeric_limits<T>::max` as the identity.

BOOLEAN VALUES: AND AND OR

$$\{\{true, false\}, \wedge, true\}$$

```
template <typename... Args>
constexpr bool all(Args&&... args) { return (... && args); }
```

$$\{\{true, false\}, \vee, false\}$$

```
template <typename... Args>
constexpr bool any(Args&&... args) { return (... || args); }
```

BOOLEAN VALUES: XOR

$$\{\{true, false\}, \oplus, false\}$$

A	B	Result
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

Note: exclusive-or on `bool` is `operator!=`

CODE INTERLUDE

Recognizing accumulation-style algorithms

CODE: THE OBVIOUS ALGORITHMS

The following algorithms are almost a dead giveaway:

- `accumulate`, `reduce`
- basically, all the algorithms in `<numeric>`
- fold expressions

`<algorithm>`: THE OTHER "USUAL SUSPECTS"

Suspect a monoid whenever you find yourself using the following algorithms:

- `all_of`, `any_of`, `none_of`
- (therefore also `find` and friends)
- `min_element`, `max_element`, `minmax_element`
- `count`, `count_if`

USEFUL REFORMULATIONS OF `accumulate`

```
template <typename InputIt, typename Size, typename T, typename BinaryOp>
constexpr auto accumulate_n(InputIt first, Size n, T init, BinaryOp op)
    -> std::pair<T, InputIt> {
  for (; n > 0; --n, ++first) {
    init = op(std::move(init), *first);
  }
  return {init, first};
}
```

The standard library has some `*_n` algorithms; it should have more.

USEFUL REFORMULATIONS OF accumulate

```
template <typename InputIt, typename T, typename BinaryOp>
constexpr T accumulate_iter(InputIt first, InputIt last, T init, BinaryOp op) {
  for (; first != last; ++first) {
    init = op(std::move(init), first);
  }
  return init;
}
```

Pass the iterator to the op ***undereferenced*****.**

MORE EXAMPLES

Because brains learn by seeing lots of variations.

STRINGS

- `string`
- `operator+` (concatenation)
- empty string

Strings form a monoid under concatenation.
The identity is the empty string.

STRING-ISH APPLICATIONS

```
std::vector<T> v{1, 2, 3, 4, 5};

std::accumulate(
    std::cbegin(v), std::cend(v), std::ref(std::cout),
    [](auto &os, const auto &elem) -> decltype(auto) { return os.get() << elem; });
```

Here, `cout` is acting like the accumulating string.

STRING-ISH APPLICATIONS

```cpp
std::string url_base = "https://example.com/?";
std::map<std::string, std::string> url_args {{"alpha", "able"},
                                             {"bravo", "baker"}};

join(std::cbegin(url_args), std::cend(url_args),
     std::back_inserter(url_base), '&',
     [] (const auto& p) {
       const auto& [key, val] = p;
       return key + '=' + val;
     });
```

We accumulate the query arguments into the url.

JOINING STRING-ISH THINGS

```
template <typename InputIt, typename OutputIt, typename T, typename Projection>
OutputIt join(InputIt first, InputIt last,
              OutputIt dest,
              T delimiter,
              Projection proj);
```

See also: `std::experimental::ostream_joiner`, `ranges::view::join`.

ANIMATIONS: A MONOIDAL THOUGHT EXPERIMENT

Consider an animation library.

What is an animation?

- a series of keyframes?
- a series of blends (curves?) between them?
- a function from time to position?

How can we compose animations?

- by pointwise operation
- by sequencing

SCHEDULES: THE FREE MONOID AGAIN

```cpp
// Schedule& Schedule::then(interval_t);

auto s = Schedule(interval::fixed{1s})
  .then(repeat::n_times{5, interval::random_exponential{2s, 2.0}})
  .then(repeat::forever{interval::fixed{30s}});

// template <typename Timer, typename Task>
// void Schedule::run(Timer, Task);
s.run(timer, task);
```

Easy to Use, Hard to Misuse: Declarative Style in C++

https://www.youtube.com/watch?v=I52uPJSoAT4

PAUSE: WHY ARE MONOIDS IMPORTANT?

Why is "abstract mathematics" important for programming?

> *"Being abstract is something profoundly different from being vague... the purpose of abstraction is not to be vague,*
> *but to create a new semantic level*
> *in which one can be absolutely precise."*

-- EWD

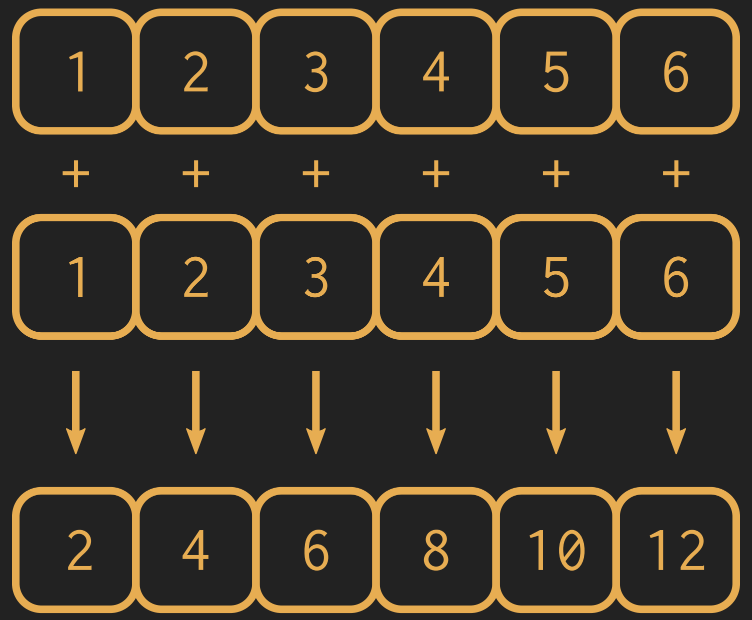
GOING FURTHER

We've seen:

- "primitive" monoids (on "number-like" things)
- the free monoid (concatenation)

Let's look at composition.

CONTAINERS



A container is a monoid on its `value_type`.

MAPS

A map is a monoid on its mapped_type.

```
std::map<std::string, int> jan_hours{{"Alice", 80},
                                     {"Bob", 90}};
std::map<std::string, int> feb_hours{{"Bob", 90},
                                     {"Charlie", 70}};

std::map<std::string, int> total_hours = ...;
// {"Alice", 80}, {"Bob", 180}, {"Charlie", 70}
```

As maps, so (pure) functions.

PRODUCT TYPES: MEMBERWISE MONOIDAL

`struct`, `pair`, `tuple`

```
using modulus_t = double;
using argument_t = double;
using polar_complex_number_t = std::pair<modulus_t, argument_t>;

using computation_t = auto (*) (int) -> int;
using profile_data_t = std::pair<computation_t, chrono::nanoseconds>;
```

SETS

(Mathematical) sets are monoidal in another way: by intersection and union.

$$\{\{sets\}, \cup, \varnothing\}$$
$$\{\{sets\}, \cap, \mathbb{U}\}$$

MONOIDAL CONFIGURATION

Let's look at another common application of several monoidal structures we've seen so far.

CONFIGURATION

- JSON objects
- configuration blobs
- sets of command-line flags
- serialization formats (e.g. Protocol buffers)

PROTOCOL BUFFERS: MONOIDS IN DISGUISE

"Normally, an encoded message would never have more than one instance of a non-repeated field. However, parsers are expected to handle the case in which they do. For numeric types and strings, if the same field appears multiple times, the parser accepts the last value it sees. For embedded message fields, the parser merges multiple instances of the same field, as if with the `Message::MergeFrom` method – that is, all singular scalar fields in the latter instance replace those in the former, singular embedded messages are merged, and repeated fields are concatenated."

https://developers.google.com/protocol-buffers/docs/encoding

PROTOCOL BUFFERS: MONOIDS IN DISGUISE

"As mentioned above, elements in a message description can be labeled optional. ... If the default value is not specified for an optional element, a type-specific default value is used instead"

https://developers.google.com/protocol-buffers/docs/proto

CODE INTERLUDE

Identity problems, arity flexibility.

VALUE TYPE PROBLEMS

Usually we would want an identity to be provided by a type's default constructor.

But sometimes, there is no good identity.

```
struct color { ... };
```

Usually for one of two reasons:

- real-world values don't have defaults
- different identities are required for different operations

IDENTITY PROBLEMS

Sometimes, an operation is closed and associative, but really has no identity.

Or, your datatype might not be able to express the identity.
(You crafted it that way for safety in other areas.)

What to do?

std::optional TO THE RESCUE

Providing a sentinel value that you can use as an identity is what `std::optional` does.

```
template <typename Operation, typename T>
auto monoid_op(std::optional<T> x, std::optional<T> y)
    -> std::optional<T> {
  if (!x) return y;
  if (!y) return x;

  return Operation{}(*x, *y);
};
```

If `T` is a semigroup, then `std::optional<T>` is a monoid.

FIRST AND LAST

`first` (and analogously `last`) is an operation on a semigroup.

```
template <typename T>
auto first(T x, T y) { return x; }
```

With `optional`, it's a monoid operation.

```
template <typename T>
auto last(std::optional<T> x, std::optional<T> y) {
  if (y) return y;
  return x;
}
```

ARITY FLEXIBILITY

You have some choices:

1. Overload an operator: get fold expressions
2. Make a special type, define operations on it, dynamic OO-style
3. Provide a traits class and generic code
4. Do something with concepts
5. Other variations…

TRAITS & CONCEPTS

```
template <typename T, typename Name> struct monoid_traits;

template <Numeric T> struct monoid_traits<T, class multiply> {
  constexpr static auto identity = [] { return T{1}; };
  constexpr static auto op = [](T a, T b) { return a * b; };
};

template <typename Name, typename... Ts>
constexpr auto fold(Ts... ts) {
    using T = std::common_type_t<Ts...>;
    using monoid = monoid_traits<T, Name>;
    T sum = monoid::identity();
    return ((sum = monoid::op(sum, ts)), ...);
};
```

MONOIDAL STATISTICS

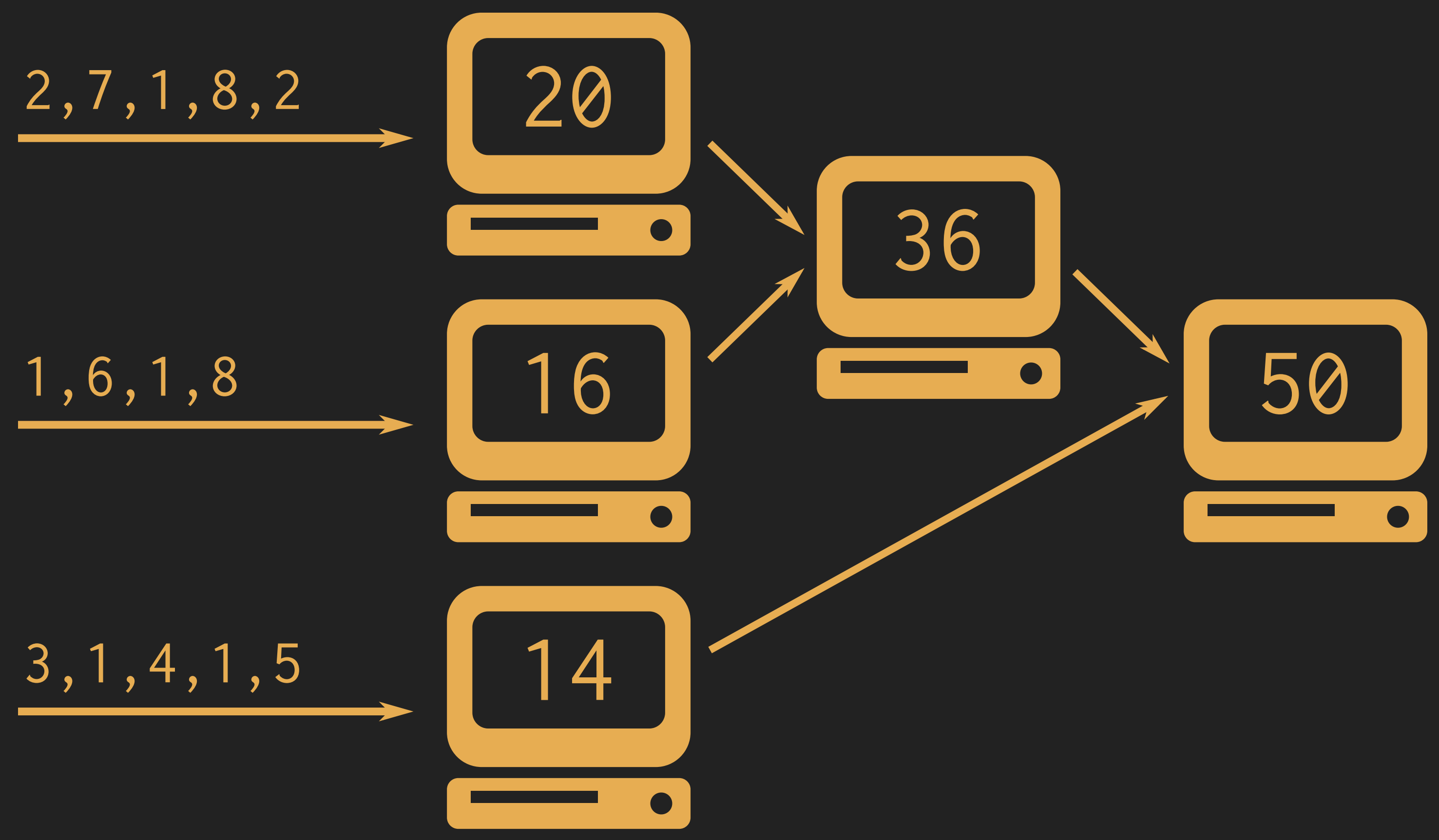
Computation of statistics is almost always monoidal.

Recognizing and exploiting monoidal properties allows us to distribute computations.

SIMPLY SUMMING (COUNTING) THINGS

Monoids are closed.

SIMPLY SUMMING (COUNTING) THINGS

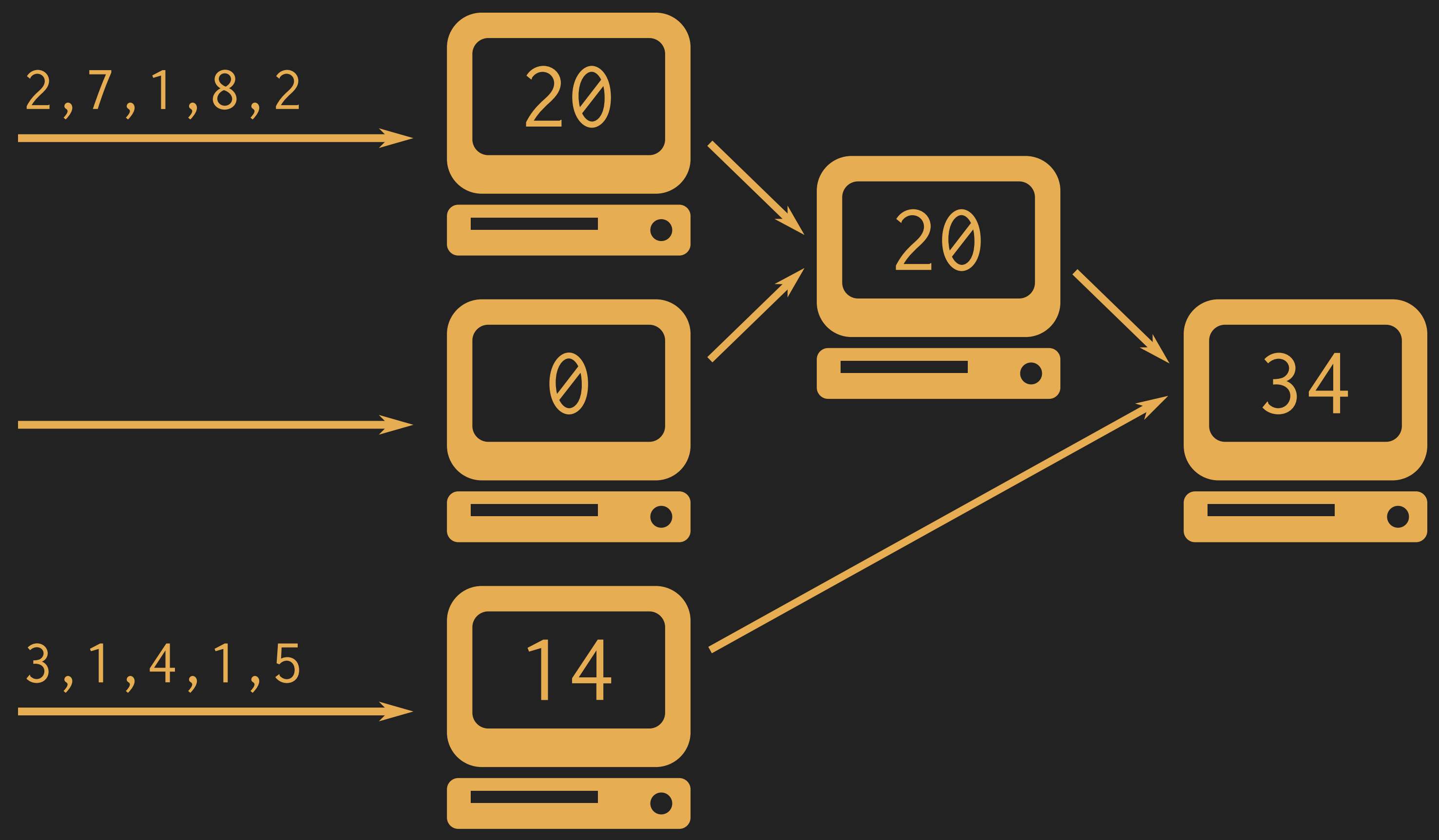


Monoids are associative.

SIMPLY SUMMING (COUNTING) THINGS

Monoids are associative.

SIMPLY SUMMING (COUNTING) THINGS



Monoids have an identity.

A FEW STATISTICAL MONOIDS

- max and min
- top N
- mean
- histogram

FANTASTIC (MONOIDAL) ALGORITHMS

Nicholas Ormrod's 2017 CppCon talk "Fantastic Algorithms and Where to Find Them".

https://www.youtube.com/watch?v=YA-nB2wjVcI

- Heavy hitters
- Reservoir sampling
- HyperLogLog

These all have monoidal structure.

HYPERLOGLOG

Intuition for HyperLogLog

- we have an ideal hash function
- we've seen N items
- the expected "inter-hash" value is $E(e) = \frac{1}{N+1}$
- therefore the expected min value is $E(e) = \frac{1}{N+1}$
- we can recover N from $\frac{1}{e} - 1$

COUNT-MIN SKETCH

Intuition for Count-Min Sketch

hash1				1		
hash2			1			
hash3					1	

insert(Alice)

COUNT-MIN SKETCH

Intuition for Count-Min Sketch

hash1			1	1		
hash2			2			
hash3	1				1	

insert(Bob)

COUNT-MIN SKETCH

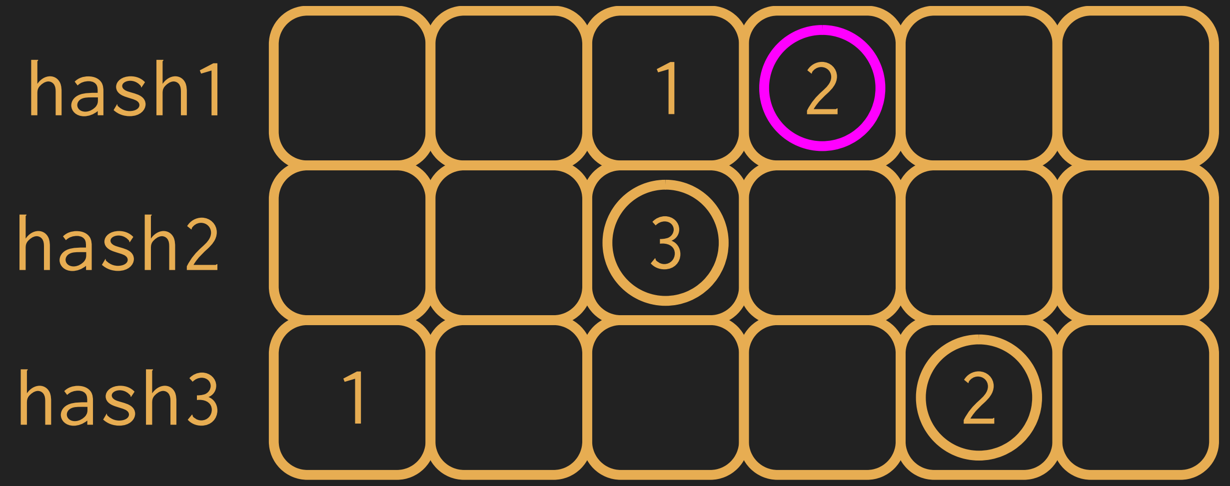
Intuition for Count-Min Sketch

hash1			1	2		
hash2			3			
hash3	1				2	

insert(Alice)

COUNT-MIN SKETCH

Intuition for Count-Min Sketch



how_many(Alice)?

MONOIDAL STRUCTURE OF DISTRIBUTED STATS

Monoids pervade distributed computations, especially statistics.

- closedness gives us bounded space
- associativity unlocks the ability to stripe across hardware/time
- identity value helps with ops

See also: Avi Bryant, *Add ALL the Things* (Strange Loop 2013)
https://www.infoq.com/presentations/abstract-algebra-analytics

INCREMENTAL COMPUTATION

Let's talk about processes evolving in time.

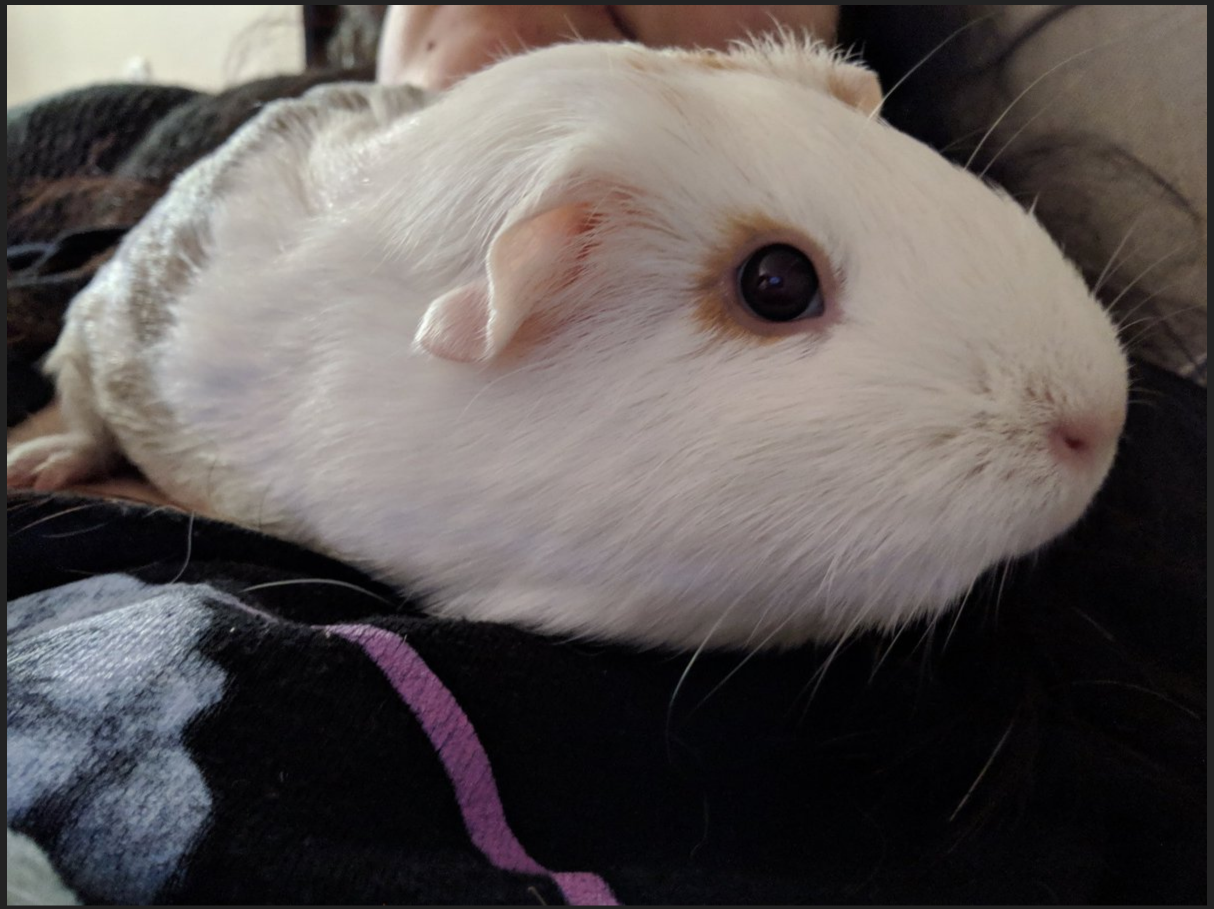
FUNCTION COMPOSITION IS A MONOID

We already saw an example of this...

```
using computation_t = auto (*) (int) -> int;
using profile_data_t = std::pair<computation_t, chrono::nanoseconds>;
```

```
using a_to_b = auto (*) (A) -> B;
using b_to_c = auto (*) (B) -> C;
```

LET'S EXAMINE `std::iota`



Ben Deane
@ben_deane

I think iota is a fine name! I named my guinea pig iota (short for std::iota of course). After all, uninitialized_default_construct_n didn't seem to suit her.

std::iota

```
template <typename ForwardIt, typename T>
void iota(ForwardIt first, ForwardIt last, T value)
{
    while(first != last) {
        *first++ = value;
        ++value;
    }
}
```

Monoidal structure lurks.

nonstd::iota

```
template <typename ForwardIt, typename T>
void iota(ForwardIt first, ForwardIt last, T value) {
  std::accumulate(first, last, value, [](const auto &so_far, auto &next) {
    next = so_far;
    return so_far + 1;
  });
}
```

The structure revealed.

nonstd::iota

```
template <typename ForwardIt, typename T, typename UnaryFunction>
void iota(ForwardIt first, ForwardIt last, T value, UnaryFunction f) {
  std::accumulate(first, last, value, [&](auto &so_far, auto &next) {
    next = so_far;
    return f(so_far);
  });
}
```

Generalization of the increment.

nonstd::iterate

```cpp
template <typename ForwardIt, typename T, typename EndoFunction>
constexpr void iterate(ForwardIt first, ForwardIt last, T init, EndoFunction f)
{
    while (first != last) {
        *first++ = init;
        init = f(std::move(init));
    }
}
// and of course iterate_n similarly

constexpr auto iota = [] (auto first, auto last, auto value) {
  iterate(first, last, value, [] (auto i) { return i + 1; });
};
```

Properly generic form.

ENDOFUNCTIONS AND PROCEDURAL GENERATION

Putting `nonstd::iterate` to work.

MAZE GENERATION

You probably know a few algorithms for maze generation.

- Recursive backtracking
- Prim's
- Kruskal's
- Aldous-Broder
- Binary tree
- Hunt-and-kill
- Wilson's
- Sidewinder
- Eller's

ELLER'S ALGORITHM

Start with a row of unlinked cells, all in different sets

Then, given a row:

- randomly link (east-west) adjacent cells from different sets, merge their sets
- randomly link south at least once from each set of cells
- any cells in the next row that were not linked from the north get new sets

To finish, link (east-west) all cells from different sets.

https://pragprog.com/book/jbmaze/mazes-for-programmers

ELLER'S ALGORITHM

DEMO

Eller's algorithm: `nonstd::iterate_n` in action.

STREAMING WITH MONOIDS

When we recognize a monoidal operation, and extract the state, we get easier incremental computation ability.

This is applicable at scale, or in the comfort of our own CPU.

MONOID HOMOMORPHISMS

"A 25-dollar term for a 5-cent concept"

(thanks Kris)

CHANGING ONE MONOID INTO ANOTHER

A *monoid homomorphism* changes one monoid into another, e.g.

- Strings form a monoid under concatenation
- Integers form a monoid under addition

`string::length` is a monoid homomorphism

- the identity is preserved (empty string has length zero)
- general structure is preserved
- the monoids are different

WE DO THIS ALL THE TIME

It's very common that we do calculations in different spaces.

- easier to think about
- easier to calculate

```
main(n){float r,i,R,I,b;for(i=-1;i<1;i+=.06,puts(""))for(r=-2;I=i,(R=r)<1;
r+=.03,putchar(n+31))for(n=0;b=I*I,26>n++&&R*R+b<4;I=2*R*I+i,R=R*R-b+r);}
```

EXAMPLE

Q. What's the best way to compute the n^{th} Fibonacci number?

A. Raise a matrix to the n^{th} power.

FIBONACCI

The Fibonacci sequence is a function:

$$\{fib_{n-1}, fib_n\} \rightarrow \{fib_n, fib_{n+1}\}$$

```
using fib = auto (*)(std::pair<int, int>) -> std::pair<int, int>;
```

ANOTHER EXAMPLE

A linear congruential PRNG is a function:

$$x_{n+1} = (ax_n + b) \mod m$$

Can we apply a similar transformation?

PRNG APPLICATIONS

```
std::linear_congruential_engine::discard(unsigned long long z);
```

"Advances the internal state by `z` times.
Equivalent to calling `operator()` `z` times and discarding the result."

"For some engines, "fast jump" algorithms are known"

LOGARITHMIC SKIPAHEAD

```
auto skip_rand = [](std::uint32_t x, int n) -> std::uint32_t {
  std::uint64_t G = x;
  std::uint64_t C = 0;
  {
    auto c = B;
    auto h = A;
    auto f = B;
    while (n > 0) {
      if (n & 1) {
        G = (G * h) % M;
        C = (C * h + f) % M;
      }
      f = (f * (h + 1)) % M;
      h = (h * h) % M;
      n >>= 1;
    }
  }
  return G + C;
};
```

FAST DISCARD

Modular exponentiation

Random Number Generation with Arbitrary Strides – **Forrest B. Brown**
https://laws.lanl.gov/vhosts/mcnp.lanl.gov/pdf_files/anl-rn-arb-stride.pdf

Also applies to other RNGs e.g.

- PCG http://www.pcg-random.org/useful-features.html#jump-ahead-and-jump-back
- xorshift https://arxiv.org/pdf/1404.0390.pdf

WHY USE A MH?

When you spot a monoid, wonder if there's a monoid homomorphism.

Maybe you can get the calculation into a different space:

- where you can do more
- where you can do things faster
- where you can think more easily

EVEN MORE ON MONOIDS

Things I don't have time to go into fully,
left as an exercise for the viewer.

When you start looking for monoids, they crop up *everywhere*.

ಠ_ಠ

```
template <typename InputIt, typename T, typename BinaryOp>
T reduce(InputIt first, InputIt last, T init, BinaryOp binary_op);
```

"The behavior is non-deterministic if binary_op is not associative *or not commutative.*"

Data-level parallelism at war with function-level parallelism...

FOLDING OVER TREE STRUCTURES

"Normal" `accumulate` has two operations:

- one for combining with identity
- one for combining with value

If we look at a tree structure as a sum type, we can extend this to:

- one for combining each type of value

std::accumulate: Exploring an Algorithmic Empire
https://www.youtube.com/watch?v=B6twozNPUoA

FOLDING OVER TREE STRUCTURES

```
struct EmptyVector {};

template <typename T>
using Vector<T> = variant<EmptyVector, pair<T, Vector<T>>>;
```

```
accumulate(first, last,
           init, // <- deal with EmptyVector
           plus) // <- deal with T
```

`nonstd::overload` **and the ability to have a recursive-lambda overload set (e.g. enabled by P0847 *deducing this*) helps.**

FOLDING OVER TREE STRUCTURES

Q: How is folding over a tree different from `std::visit`?

A: The same way `accumulate` is different from `for_each`.

```
const auto sum = std::accumulate(first, last, 0);

auto sum = 0;
std::for_each(first, last, [&] (auto value) { sum += value; };
```

FUTURES AS MONOIDS

No, not as monads. Not today.

Futures form a monoid with the "race" operation. (`when_any`)

- (A 'race' B) 'race' C == A 'race' (B 'race' C)
- the identity is `never` (the future that never completes)

PARSERS AS MONOIDS

You remember:

> *"A parser for things is a function from strings to lists of pairs of things and strings."*

-- Fritz Ruehr, Dr. Seuss on Parser Monads

Parsers form a monoid under alternation.

- (A | B) | C == A | (B | C)
- the identity is the parser that always fails

constexpr ALL the things!

https://www.youtube.com/watch?v=HMB9oXFobJc

TRAINING SETS AS MONOIDS

You have a large set of data to train on.

Maybe you have a monoid.

Train on large set => produce distribution.

Train on incremental data => produce distribution.

Can you combine the distributions monoidally?

https://izbicki.me/blog/gausian-distributions-are-monoids.html

INCREMENTAL REGULAR EXPRESSION MATCHING WITH MONOIDS

You have:

- a regular expression
- a string to match

Perform the match once, then edit the string.

How expensive is performing a second match?

http://blog.sigfpe.com/2009/01/fast-incremental-regular-expression.html

THE MONOID GAME

or, how to ~~troll~~ engage with Haskell programmers

You: "X is a monoid!"

Haskeller, interested: "Ooh, you mean in the sense of [FP stuff]?"

You: "Up to isomorphism."

This works because *almost everything* is a monoid under some interpretation.

FINAL THOUGHTS

- thinking about structure helps to separate control flow from logic
- monoids are a ubiquitous pattern for doing that
- try to think beyond numerics
- added benefit: distributed and/or incremental computation

> *"Discovery consists of seeing what everybody has seen, and thinking what nobody has thought."*

-- Albert Szent-Györgyi (Nobel Laureate in Medicine, 1937)