Deriving a Generic Algorithm

Lecture 9

Hartmut Kaiser

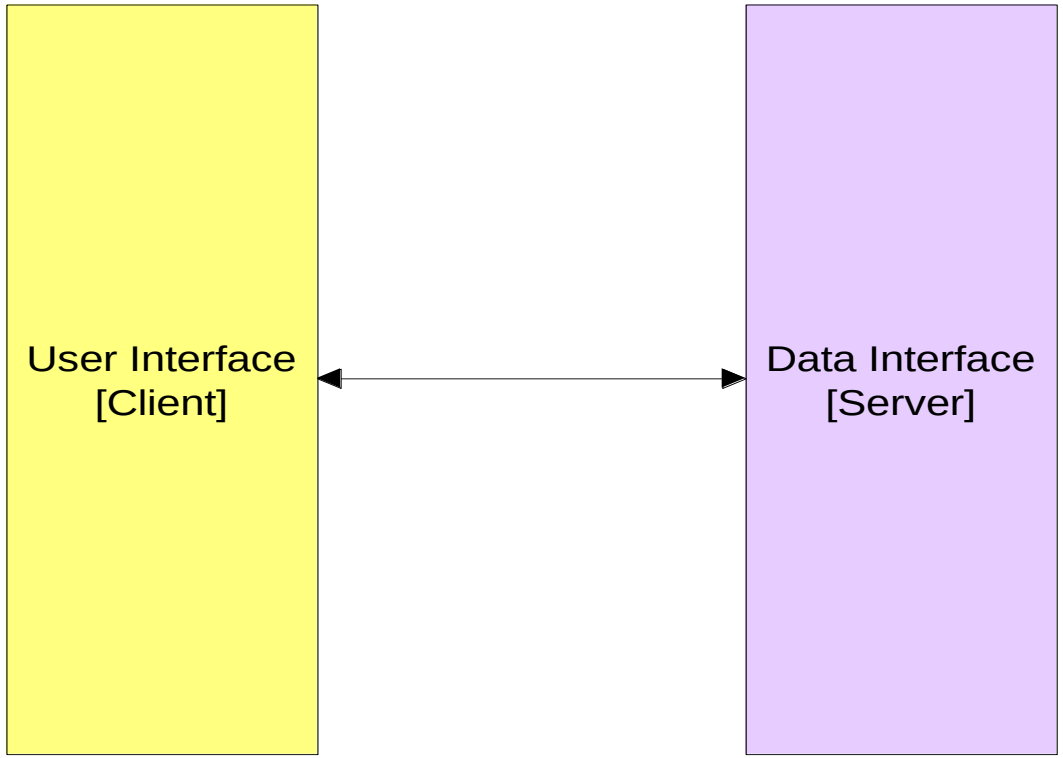
https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

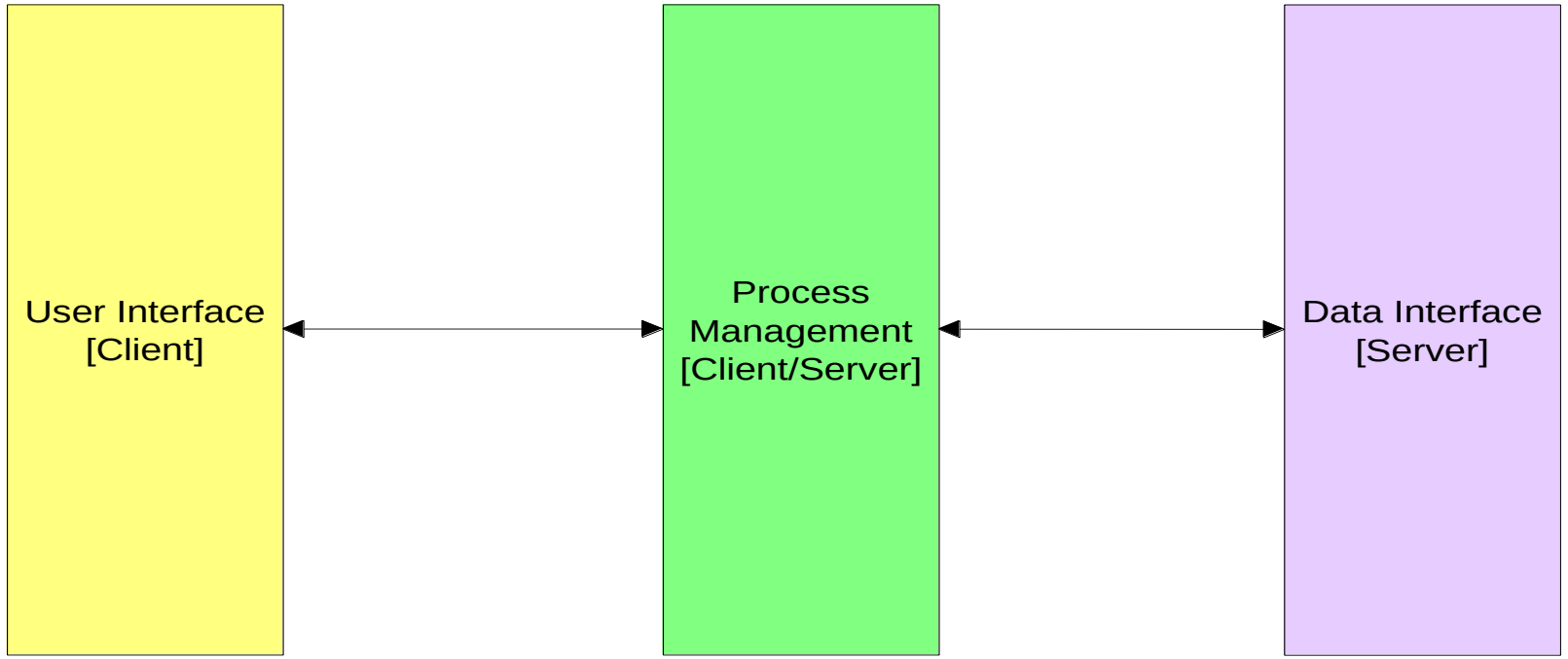
Client/Server

- A network architecture in which each computer or process on the network is either a client or a server
  - Servers are processes dedicated to managing resources, such as disk drives (file servers), printers (print servers), network traffic (network servers), or computational services
    - The server “guards” access to the important resources that it manages
  - Clients are programs, which are run by users or specialized applications. Clients rely on servers for resources, such as files, devices, data, and computations (e.g., processing power)
    - Clients connect to the server

Client Server Abstraction: 2-Tier Architecture



Client Server Abstraction: 3-tier Architecture



- Process Management acts as a server to the User Interface client
- It acts as a client to the Data Interface Server

Examples of Client/Server

- Multiplayer videogames:
  - Clients absolutely cannot be trusted to change game state directly
  - Server validates actions and ensures rules are followed
  - Doesn't help with client side exploits
- X11 windowing system
  - Allows a windowing application to be separated from a "view"
  - Streams drawing commands over network
  - Allows low-latency screen sharing, but still being replaced
- Modern single-page web applications
  - Facebook, Twitter, Youtube, etc.
  - Client runs in your browser, the server runs in the cloud

A Note on Using Web Servers

- If you plan on developing a web app, don't rewrite a webserver yourself
- Consult your mentor TA on options
  - The absolute easiest way to have a web-based server is for it to be a (fast) CGI application
  - Install an http server on your host
    - Nginx is good one
    - Apache is widely supported
  - Configure it to call your program
  - Your server application will return either HTML or JSON

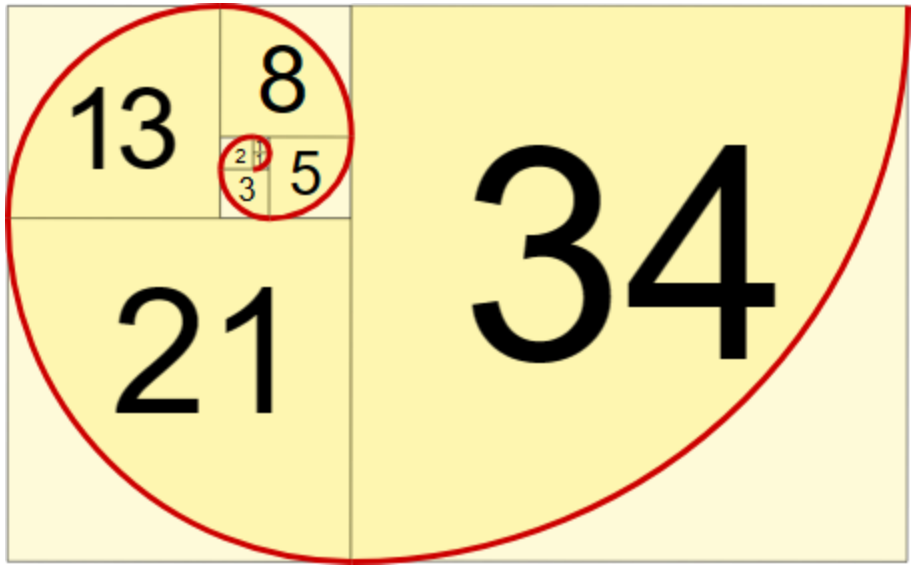
Fibonacci Numbers

Introduction

- We will take the Egyptian Multiplication algorithm and generalize it in order to apply it to a wide variety of problems beyond simple arithmetic
- Good piece of code requires (at least) two steps
  - Write the algorithm
  - Find out what type requirements the algorithm imposes on its arguments
- Why generalize?
  - Holy grail of programming is to reuse code
  - You might not even know who is going to use your code for what application in the future

Fibonacci Sequence

- Every next number is computed by adding the previous two
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



Fibonacci Sequence

Fibonacci Sequence

```
int fib(int n)
{
    if (n == 0) return 0;
    std::pair<int, int> v = {0, 1};
    for (int i = 0; i < n; ++i)
    {
        v = {v.second, v.first + v.second};
    }
    return v.first;
}

int main()
{
    std::cout << fib(10) << "\n";
}
```

Complexity: $O(n)$

Is that the best we can do?

Egyptian Multiplication

Algorithm Requirements

- This is where we stopped:

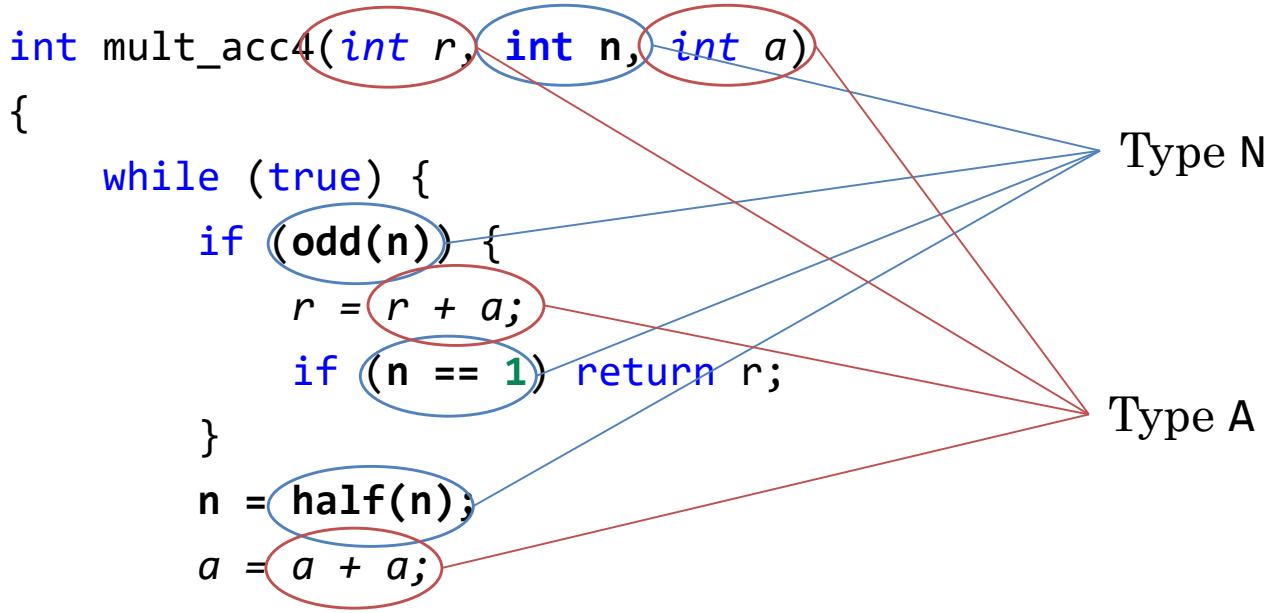
```
int mult_acc4(int r, int n, int a)
{
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Algorithm Requirements

- Note the operations on **n** and *r* are orthogonal and non-overlapping
- The requirements on **n** and *r* are different and could be represented by different types
  - So far we have been using the same type `int` for both
- Let's call type types `N` and `A`
  - `A` must be 'addable' (support `operator+()`)
  - `N` must be checkable for odd-ness, compared with `1`, and must support division by `2`

Algorithm Requirements

- This is where we stopped:



Algorithm Requirements

- More generic form of the algorithm:

```
template <typename A, typename N>
A multiply_accumulate(A r, N n, A a)
{
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

- Allows to figure out requirements for `N` and `A` separately

Requirements on A

- Syntactical requirements:
  - Can be passed by value (must have a copy constructor)
  - Can be assigned to another instance (must have copy assignment, i.e. `operator=()`)
  - Can be added (i.e. must have `operator+()`)
- Semantic requirements:
  - `operator+()` must be associative:

$$A(T) \Rightarrow \forall a, b, c \in T\colon\ a + (b + c) = (a + b) + c$$

  - if a type T is an A (i.e. is associative), then for any values a, b, and c in T, the following holds…

Requirements on A: Clarification

- While adding integers is associative in theory, on computers this might not be true
  - Consider: (x + y) + z, for large x, y, and a negative z
    - x + y might overflow
    - Yields possibly different result from x + (y + z)
  - operator+() on computers usually is a *partial function*
    - A partial function's domain of definition is a subset of the direct product of the types of its input
    - It is defined only for a subset of all possible combinations of values of its input
- Let's clarify requirements:
  - The requirement holds only in the *domain of definition* of the function, i.e. for the set of values for which the function is defined

Requirements on A: regularity

- More syntactical requirements
  - Copy construction and copy assignment should make the copy **equal** to the original
    - So we need to have means of testing for equality: `operator==()` and `operator!=()`
- More semantic requirements
  - Equational reasoning must be applied:
    - Inequality must be the negation of equality:

$$(a \neq b) \Leftrightarrow \neg(a = b)$$

    - Equality is reflexive, symmetric, and transitive (i.e. *equivalence*):

$$a = a$$

$$a = b \Leftrightarrow b = a$$

$$(a = b) \wedge (b = c) \Leftrightarrow (a = c)$$

    - Equality implies substitutability:

$$\text{for any function f on T, a} = b \Rightarrow f(a) = f(b)$$

Requirements on `A`

- Shorthand requirements description
  - Must be regular type
  - Must provide associative `operator+()`
- Going back to math:
  - `A` must be a *non-commutative additive semigroup*
- Examples:
  - Positive even integers
  - Negative integers
  - Real numbers
  - Polynomials
  - Planar (2D) vectors
  - Boolean functions
  - Line segments
  - Etc.

Algorithm Requirements

- Now we are able to write:

```
template <typename T>
concept Addable = requires(T a, T b) { a + b; };  // "a+b will compile"

template <typename T>
concept NoncommutativeAdditiveSemigroup = std::regular<T> && Addable<T>;
    // there is no way to express associativity in a concept
```

Algorithm Requirements

- Now we are able to write:

```
template <typename A, typename N>
    requires(NoncommutativeAdditiveSemigroup<A>)
A multiply_accumulate(A r, N n, A a) {
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Algorithm Requirements

- Now we are able to write:

```
template <NoncommutativeAdditiveSemigroup A, typename N>
A multiply_accumulate(A r, N n, A a) {
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Requirements on `N`

- Syntactic requirements
  - Has to be `regular` as well
  - Must implement:
    - `half`
    - `odd`
    - `== 0`
    - `== 1`
- Semantic requirements
  - $even(n) \Rightarrow half(n) + half(n) = n$
  - $odd(n) \Rightarrow even(n-1)$
  - $odd(n) \Rightarrow half(n-1) = half(n)$
  - $n \leq 1 \ \vee \ half(n) = 1 \ \vee \ half\big(half(n)\big) = 1 \vee \cdots$

`uint8_t`, `int8_t`, `uint64_t`, etc.

i.e. all integral types

Algorithm Requirements

- Now finally we are able to write a fully generic algorithm:

```
template <typename T>
concept Addable = requires(T a, T b) { a + b; };  // "a+b will compile"

template <typename T>
concept NoncommutativeAdditiveSemigroup = std::regular<T> && Addable<T>;

template <typename T>
concept Integer = std::integral<T>;
```

Algorithm Requirements

- Now finally we are able to write a fully generic algorithm:

```
template <NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_accumulate_semigroup(A r, N n, A a) {  // precondition: n >= 0
    if (n == 0) return r;
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Final multiply Algorithm

```
template <NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_semigroup(N n, A a)
{
    // precondition: n > 0
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) {
        return a;
    }
    return multiply_accumulate_semigroup(a, half(n - 1), a + a);
}
```

Further Generalization: Monoid

- Let's lift the precondition that `n` should be strictly greater than zero
- What should `multiply` return if n == 0?
  - The value that doesn't change the result if the semigroup's operation is applied
  - In our case it should return the *additive identity*
- But semigroups are not required to have an identity element:

$$x \circ e = e \circ x = x$$

- As a result we could change the requirements for the type to be a `Monoid`
  - I.e. a `Semigroup` that additionally has an identity element
  - For non-commutative additive monoid, the identity element is called '0':

$$x + 0 = 0 + x = x$$

Further Generalization: Monoid

```
template <typename T>
concept HasAdditiveIdentity = requires(T) { T(0); };

template <typename T>
concept NoncommutativeAdditiveMonoid =
    NoncommutativeAdditiveSemigroup<T> && HasAdditiveIdentity<T>;
```

Further Generalization: Monoid

```
template <NoncommutativeAdditiveMonoid A, Integer N>
A multiply_monoid(N n, A a)
{
    // precondition: n >= 0
    if (n == 0) return A(0);
    return multiply_semigroup(n, a);
}
```

Further Generalization: Group

- Let's lift the precondition that `n` should be non-negative
  - So multiply by negative must makes sense
  - The type must support an *inverse operation*
- Monoids do not require such an operation to be supported
- So we require a `Group`, that in addition to a `Monoid` requires a inverse operation:

$$x \circ x^{-1} = x^{-1} \circ x = e$$

- In our case:

$$x + -x = -x + x = 0$$

Further Generalization: Monoid

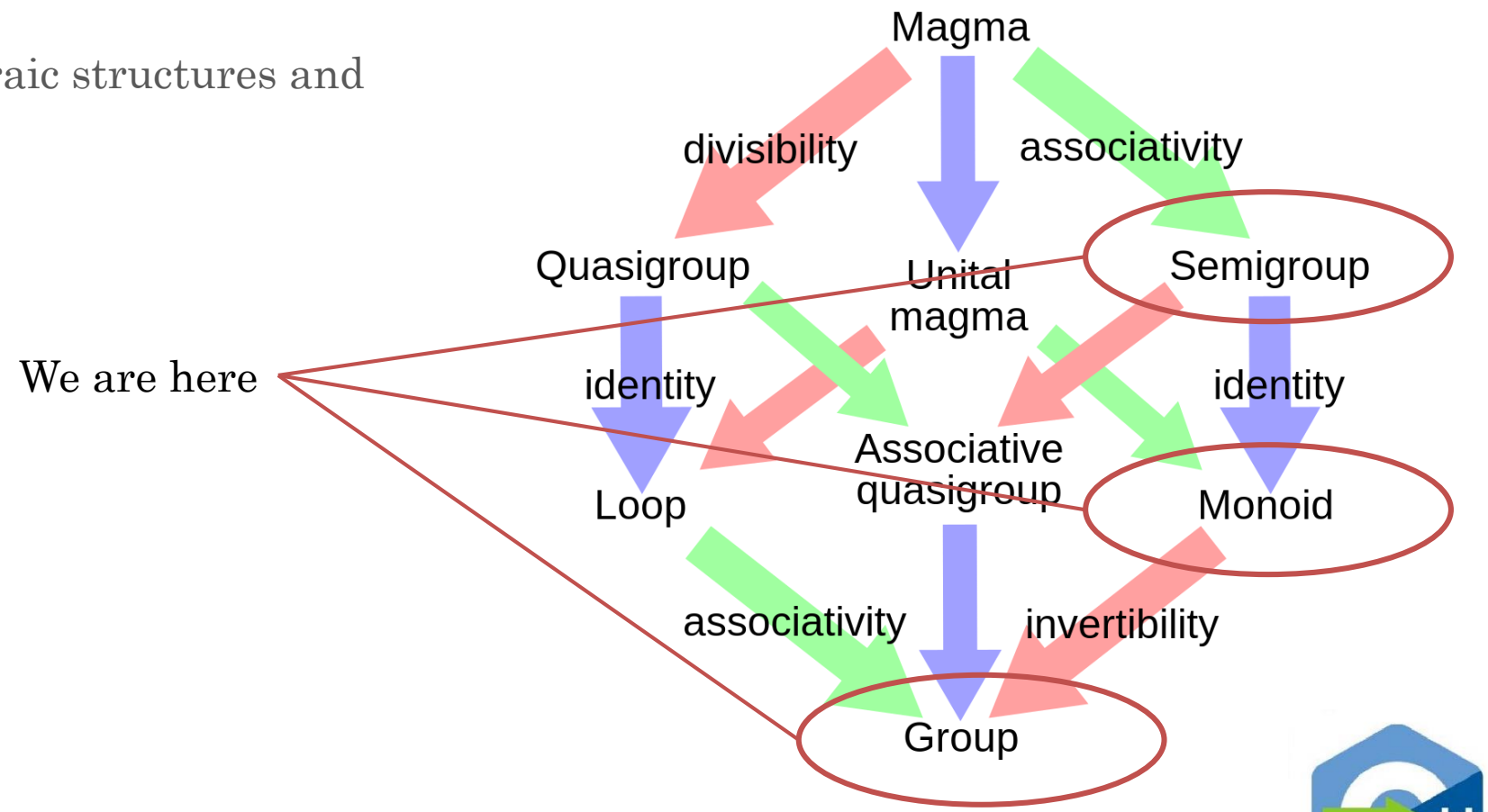
```
template <typename T>
concept AdditiveIsInvertible = requires(T a) { -a; };

template <typename T>
concept NoncommutativeAdditiveGroup =
    NoncommutativeAdditiveMonoid<T> && AdditiveIsInvertible<T>;
```

Further Generalization: Group

```
template <NoncommutativeAdditiveGroup A, Integer N>
A multiply_group(N n, A a)
{
    // precondition: none
    if (n < 0) {
        n = -n;
        a = -a;
    }
    return multiply_monoid(n, a);
}
```

Algebraic Structures

- Just FYI
  - Various algebraic structures and their relation



Turning Multiply into Power

- Now we have generalized a very efficient multiplication algorithm for types that conform to the requirements of
  - `SemiGroup`, `Monoid`, and `Group`
- If we replace `operator+()` with `operator*()` (i.e. replacing *doubling* with *squaring*), the existing algorithm can be used to calculate the power
  - Instead $n \cdot a$, we compute a^n
  - Analogous code, just for `MultiplicativeSemigroup`, `MultiplicativeMonoid`, and `MultiplicativeGroup`
- As the identity element we use `A(1)`
- As the inverse we use `A(1) / a`

Generalizing the Operation

- We have two versions of the code, both almost identical
- There are many types that are semigroups, monoids, or groups
  - Examples:
    - Integer multiplication mod 7
    - Matrix multiplication
- Let's avoid having to create yet another copy of almost the same code for each of those
- Let's pass the required operation as an additional argument

Generalizing the Operation

```
template <Regular A, Integer N, SemigroupOperation Op>
    requires(Domain<Op, A>)    // domain of Op must be A
A power_accumulate_semigroup(A r, A a, N n, Op op)
{
    // precondition: n > 0
    while (true) {
        if (odd(n)) {
            r = op(r, a);      // was: r + a
            if (n == 1) return r;
        }
        n = half(n);
        a = op(a, a);          // was: a + a
    }
}
```

Generalizing the Operation

```
// check whether the domain of a given operation is the same as its argument
template <typename Op, typename A>
concept Domain = std::same_as<A, std::invoke_result_t<Op, A, A>>;

// we can't check whether Op is associative, assume true
template <typename Op>
concept SemigroupOperation = true;
```

Generalizing the Operation

- For any `SemiGroup` we can now write:

```
template <Regular A, Integer N, SemigroupOperation Op>
    requires(Domain<Op, A>)
A power_semigroup(A a, N n, Op op)
{
    // precondition: n > 0
    while (!odd(n)) {
        a = op(a, a);     // was: a + a
        n = half(n);
    }
    if (n == 1) return a;
    return power_accumulate_semigroup(a, op(a, a), half(n - 1), op);
}
```

Generalizing the Operation

```
template <typename Op>
concept OperationHasIdentityElement = requires(Op op) { identity_element(op); };

template <typename Op>
concept MonoidOperation =
    SemigroupOperation<Op> && OperationHasIdentityElement<Op>;
```

Generalizing the Operation

- For any `Monoid` we can now write:

```
template <Regular A, Integer N, MonoidOperation Op>
    requires(Domain<Op, A>)
A power_monoid(A a, N n, Op op)
{
    // precondition: n >= 0
    if (n == 0)
        return identity_element(op);    // was: A(0)
    return power_semigroup(a, n, op);
}
```

- Note that we extract the identity element from the operation

Extracting the Identity Element

```
// The additive identity is zero
template <NoncommutativeAdditiveMonoid T>
T identity_element(std::plus<T>)
{
    return T(0);
}

// The multiplicative identity is one
template <MultiplicativeMonoid T>
T identity_element(std::multiplies<T>)
{
    return T(1);
}
```

Generalizing the Operation

```
template <typename Op>
concept OperationHasInverse = requires(Op op) { inverse_operation(op); };

template <typename Op>
concept GroupOperation = MonoidOperation<Op> && OperationHasInverse<Op>;
```

Generalizing the Operation

- For any `Group` we can now write:

```
template <Regular A, Integer N, GroupOperation Op>
    requires(Domain<Op, A>)
A power_group(A a, N n, Op op)
{
    // precondition: none
    if (n < 0) {
        n = -n;
        a = inverse_operation(op)(a);     // was: -a
    }
    return power_monoid(a, n, op);
}
```

- Note that the operation returned from `inverse_operation` is immediately invoked

Extracting the Inverse Operation

```
// The additive inverse operation is negate
template <NoncommutativeAdditiveGroup T>
auto inverse_operation(std::plus<T>)
{
    return std::negate<T>();     // create new instance of negate operation
}

template <MultiplicativeGroup T>
struct reciprocal
{
    T operator()(T x) const {  return T(1) / x; }
};

// the multiplicative inverse is reciprocal
template <MultiplicativeGroup T>
auto inverse_operation(std::multiplies<T>)
{
    return reciprocal<T>();     // create new instance of reciprocal operation
}
```

Fibonacci Numbers

Fibonacci Sequence

- We can compute the nth Fibonacci number with complexity $O(\log n)$
  - But how?
- The next Fibonacci number can be computed from the previous one by:

$$\begin{bmatrix} v_{i+1} \\ v_i \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_i \\ v_{i-1} \end{bmatrix}$$

- Then the nth Fibonacci number can be computed from:

$$\begin{bmatrix} v_n \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

- We need to raise the matrix $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ to a power – you see where this is going!

Fibonacci Sequence

- How can we be sure that we can use the power algorithm we developed?
- It is known that
  - Matrix multiplication is associative
  - Square matrices form a multiplicative `SemiGroup`
- We showed that our power algorithm is usable with any `SemiGroup` that defines an associative operation
- We can be sure that our power algorithm gives correct results for raising matrices to a given integral power

q.e.d.

The matrix Type

- Take `singleton<T>` and rename it, add members as needed:

```
template <typename T>
struct matrix
{
    T x11, x12;
    T x21, x22;
    // ...
};
```

The matrix Type

- Ensure that `matrix<T>` is `Regular`:

```
friend bool operator==(matrix const& lhs, matrix const& rhs)
{
    return lhs.x11 == rhs.x11 &&
           lhs.x12 == rhs.x12 &&
           lhs.x21 == rhs.x21 &&
           lhs.x22 == rhs.x22;
}

friend bool operator!=(matrix const& lhs, matrix const& rhs)
{
    return !(lhs == rhs);
}
```

The matrix Type

- Adding the required support for multiplication

```
friend matrix operator*(matrix const& lhs, matrix const& rhs)
{
    return matrix{
        lhs.x11 * rhs.x11 + lhs.x12 * rhs.x21,
        lhs.x11 * rhs.x12 + lhs.x12 * rhs.x22,
        lhs.x21 * rhs.x11 + lhs.x22 * rhs.x21,
        lhs.x21 * rhs.x12 + lhs.x22 * rhs.x22
    };
}
```

Final Fibonacci Code

```
int fib(int n)
{
    if (n < 2)
        return n;

    auto result =
        power_semigroup(matrix{1, 1, 1, 0}, n - 1, std::multiplies<matrix<int>>());

    return result.x11;
}
```

Fibonacci Instrumented

```
long fib_sequential(int n)
{
    if (n == 0)
        return 0;

    std::pair<instrumented<long>, instrumented<long>> v = {0, 1};
    for (int i = 0; i < n - 1; i++)
    {
        v = {v.second, v.first + v.second};
    }
    return v.first.value;
}
```

Fibonacci Instrumented

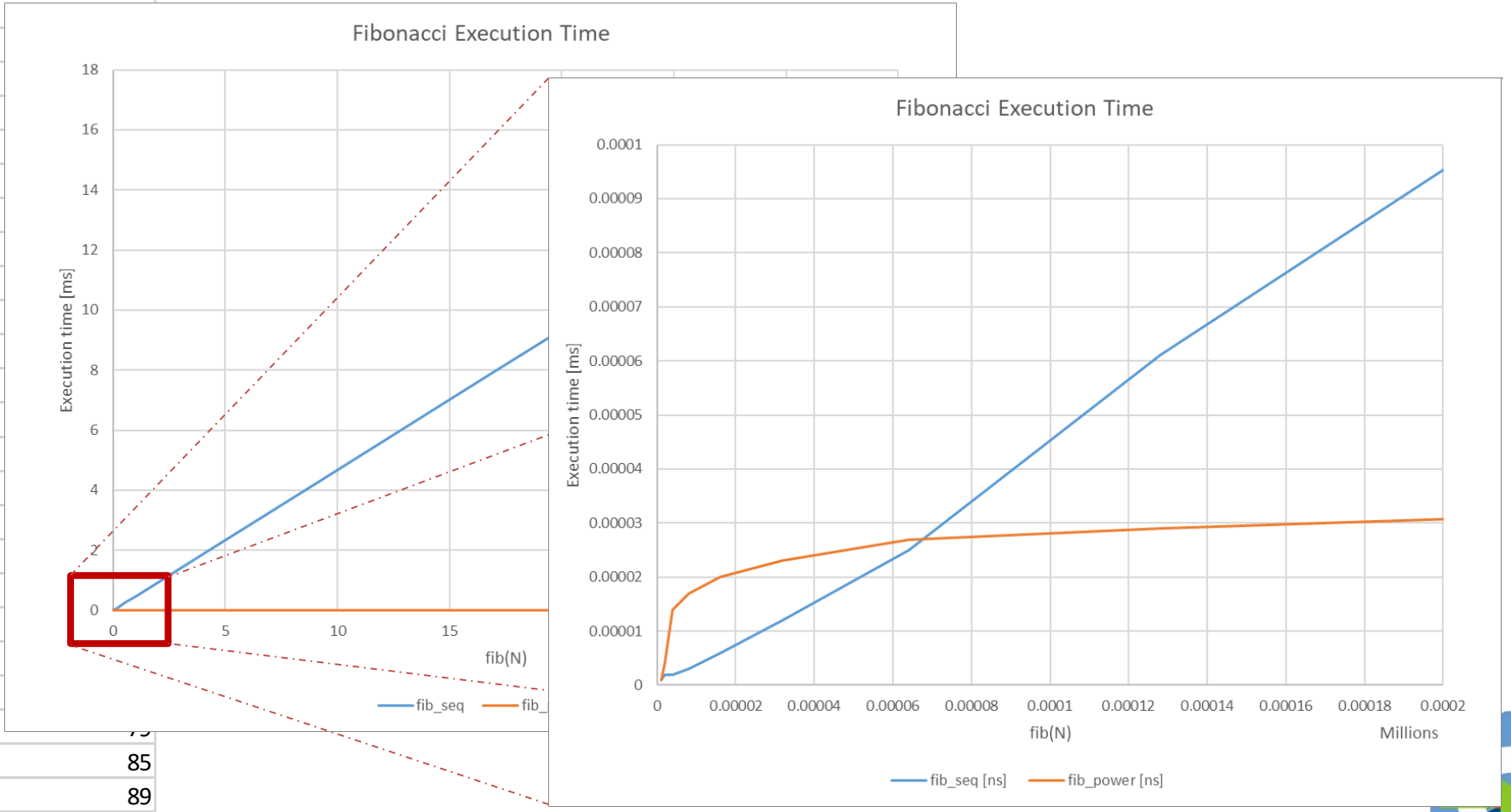
```
long fib(int n)
{
    if (n < 2)
        return n;

    auto result = power_semigroup(
        matrix<instrumented<long>>{1, 1, 1, 0}, n - 1,
        std::multiplies<matrix<instrumented<long>>>());

    return result.x11.value;
}
```

Fibonacci Instrumented

	fib_sequential		fib	
n	plus	multiply	plus	multiply
16	15	0	24	48
32	31	0	32	64
64	63	0	40	80
128	127	0	48	96
256	255	0	56	112
512	511	0	64	128
1024	1023	0	72	144
2048	2047	0	80	160
4096	4095	0	88	176
8192	8191	0	96	192
16384	16383	0	104	208
32768	32767	0	112	224
65536	65535	0	120	240
131072	131071	0	128	256
262144	262143	0	136	272
524288	524287	0	144	288
1048576	1048575	0	152	304
2097152	2097151	0	160	320
4194304	4194303	0	168	336
8388608	8388607	0	176	352
16777216	16777215	0	184	368
33554432	33554431	0	192	384

Fibonacci Benchmark

N	fib_seq [ns]	fib_power [ns]
1	1	
2	2	
4	2	
8	3	
16	6	
32	12	
64	25	
128	61	
256	122	
512	246	
1024	497	
2048	975	
4096	1973	
8192	4098	
16384	7885	
32768	16100	
65536	32071	
131072	63826	
262144	124946	
524288	253959	
1048576	496038	
2097152	985690	
4194304	1960740	85
8388608	3918000	89
16777216	7856910	93
33554432	15622600	100



Conclusions

- Doing more is sometimes faster than doing less
- Understanding the Big-O complexity characteristics of algorithms is important
- Understanding the type requirements for algorithms enables code reuse

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP