Get some Fibers in your diet!

Mikola Lysenko

University of Wisconsin-Madison,
Spatial Automation Lab

Outline

- Motivation
- What are Fibers?
- How to implement them?
- Why put them in the core?

A Common Problem

- Say you write a binary tree (ex. Parse tree)
- You want to provide some traversal operations
  - Post-Order, Pre-Order, In-Order
- Also want to support algorithms on top of streams
  - Sum, sort, min, max, etc.
- How do we do this?

Iterators

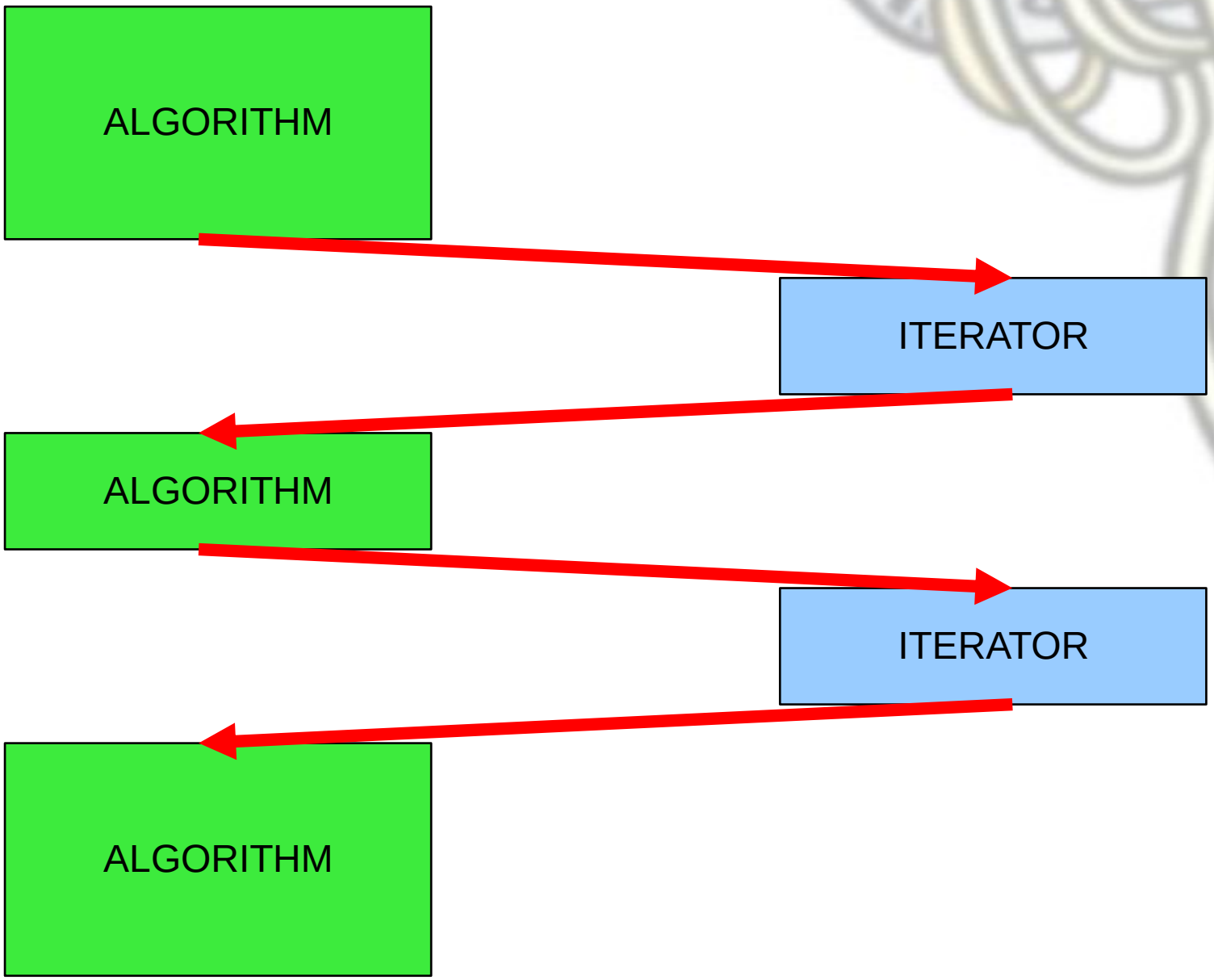
- Most common solution in today's libraries
  - C++ STL [SGI]
  - Java's Standard Library [Sun]
- Nice interface for manipulating data
- Algorithms are (relatively) easy to write

Interfacing with Iterators

```
//Convert to prefix
foreach(x ; tree.preorder)
        Stdout.formatln(x);

//Split stream and swap order
auto iter = tree.inorder;
for(int i=0; i<10; i++)
{
        auto x = iter();
        auto y = iter();
        Stdout.formatln(y);
        Stdout.formatln(x);
}
```

Iterator Control Flow



Implementing Iterators

- At best, it is tricky
  - Pre-Order/Post-Order require parent pointers
- At worst, it is really ugly
  - In-Order is tough
  - Can re-do traversal each iteration
    - O(n^2) traversal cost
  - Can store extra stack of nodes, but gets ugly
- Clearly iterators have some issues!

Visitors

- Writing the implementation code for iterators is a mess, surely there is a better way?
- Can use *“iteration by first class functions”*
- Also known as the *Visitor* pattern [GoF94]
- Same idea as `foreach/opApply`

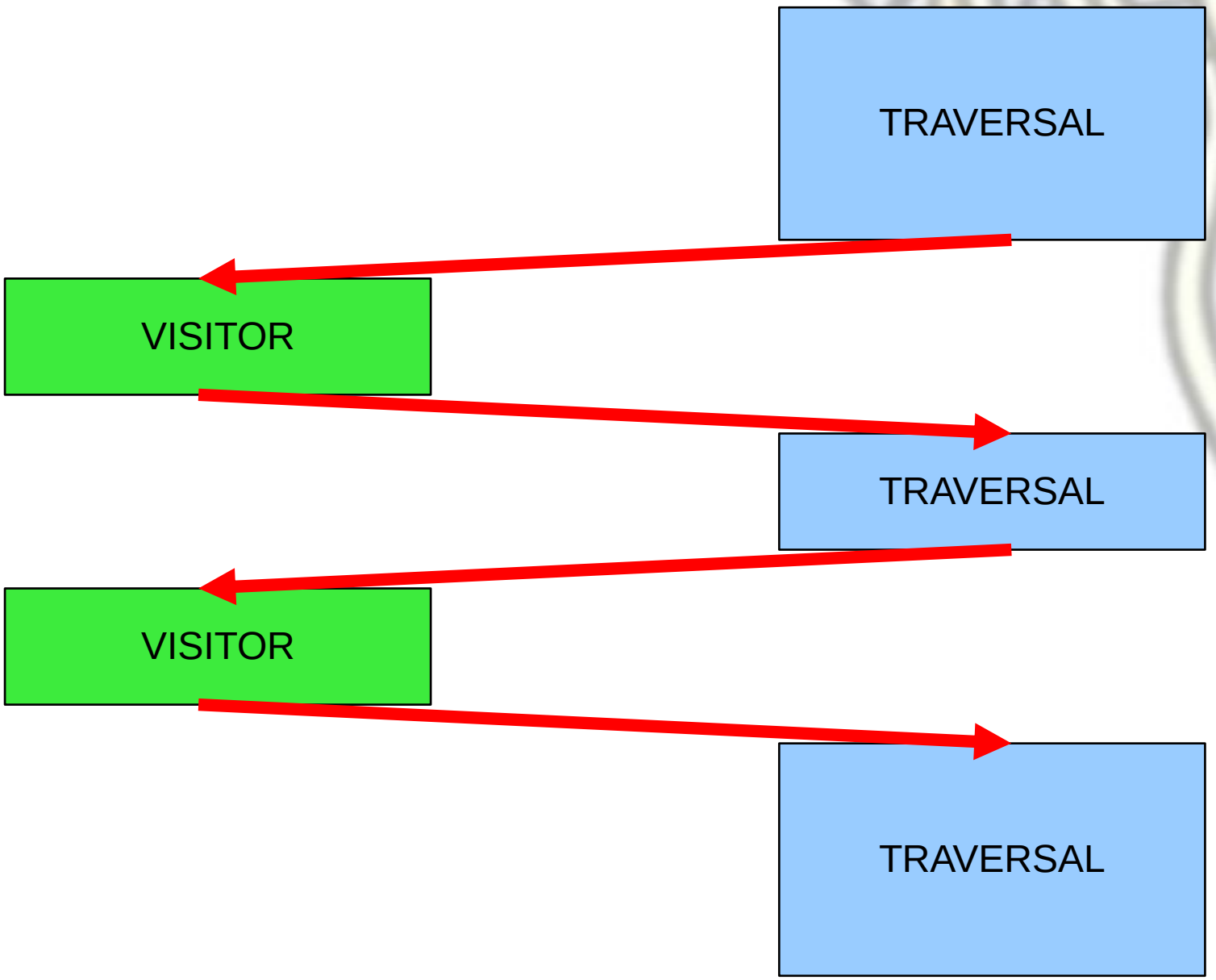
Visitor In-Order Traversal

```
//In-order tree traversal
void traverse(void delegate(Node) visitor)
{
        if(left !is null)
                left.traverse(visitor);

        visitor(this);

        if(right !is null)
                right.traverse(visitor);
}
```

Visitor Control Flow



The Problem with Visitors

- The user code is not so simple
  - How do we do 'split'?
- Must translate everything to 'map/reduce'
- This requires complex state machines
- Complex Code => Buggy Code

An Overview

- Iterators:
  - Easy algorithms, Hard Traversals
  - Algorithm controls stack
- Visitors:
  - Hard algorithms, Easy Traversals
  - Traversal controls stack
- Can we have it both ways?
  - *Yes!*

The Solution: Fibers

- Give both sides a stack!
- Algorithm code looks like an iterator
- Traversal code looks like a visitor
- Result is a 'generator' [Liskov76]
- What does this look like?

Generator Example (hypothetical)

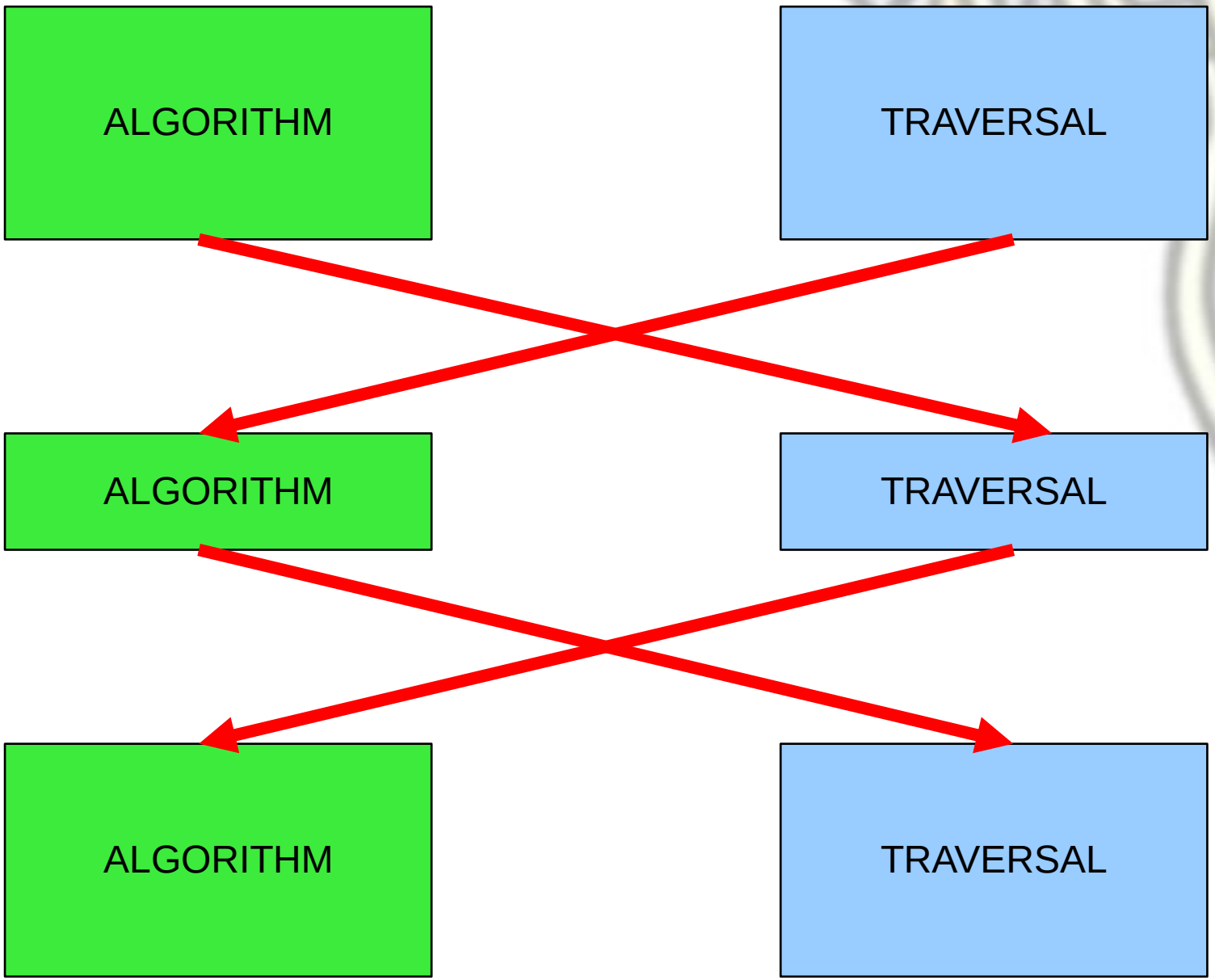
```
//Split nodes
void split()
{
        for(int i=0; i<100; i++)
        {
                auto a = traverse();
                auto b = traverse();
                Stdout.formatln(b);
                Stdout.formatln(a);
        }
}
```

```
//In-order tree traversal
Node traverse()
{
        if(left !is null)
                left.traverse();

        yield(this);

        if(right !is null)
                right.traverse();
}
```

Generator Control Flow



Generators Continued

- Generators have all benefits of iterators/visitors
  - Trivial to turn any visitor/iterator into a generator
- “Can have our cake and eat it too.”
- How do we implement them?
  - Coroutines!

What is a Coroutine?

- COncurrent ROUTINES vs. SUBROUTINES
- Functions with independent stacks
- Example:

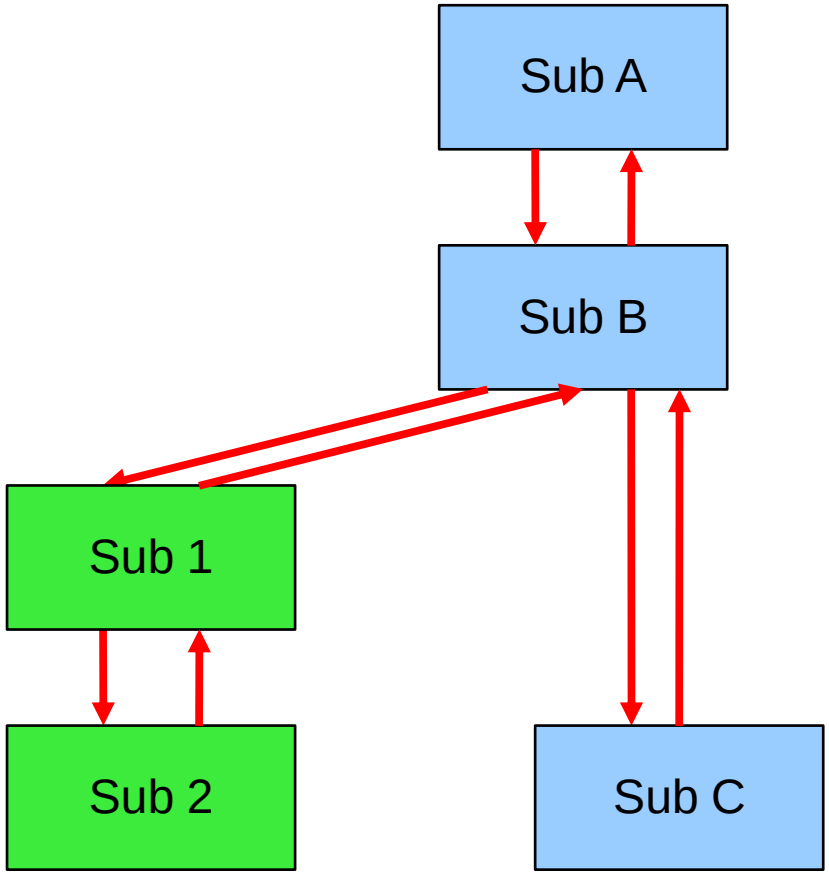
```
auto a = new Fiber({
    Stdout.formatln("a");
    Fiber.yield();
    Stdout.formatln("b");
});

a.call();
Stdout.formatln("c");
a.call();
```

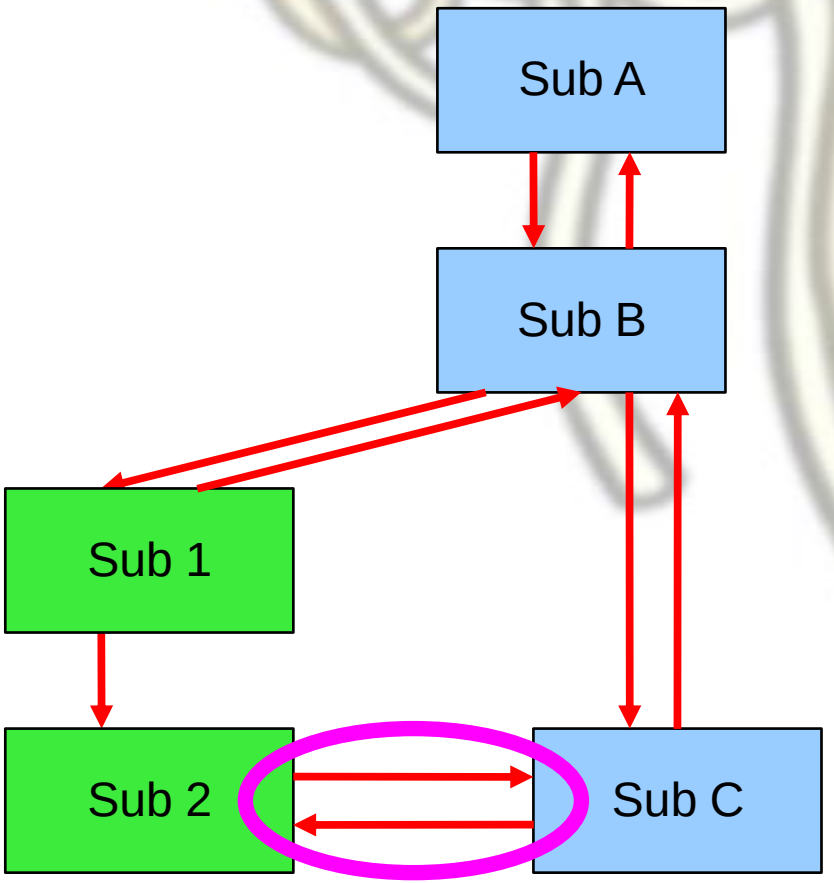
Will Print:

a
c
b

Coroutine Diagram



Subroutines



Coroutines

Previous Work: Coroutines

- Term due to [Knuth68]
  - Has been rediscovered many times
- Really old [Landin65]
  - Used on UNIVAC!!!
- Necessary for several paradigms
  - Actor Model Concurrency
  - Object Oriented Programming (SIMULA style)
  - Process Based Programming

Coroutine Awareness

- Uncommon feature
- Only 1 of top 10 TIOBE languages
  - Python (as of 2007)
- ...and even then limited users...
  - Only 7000 hits for 'stackless' out of 1.4 million Python sources [Google Code Search]
- Must educate users!

Related D Projects

- StackThreads
  - Basis for Tango's Fibers
- Daniel Keep's Coroutine Library
- PyD – Kirk McDonald
  - Python bindings for D
- MiniD – Jarrett Billingsley
  - D-like scripting language with coroutine support

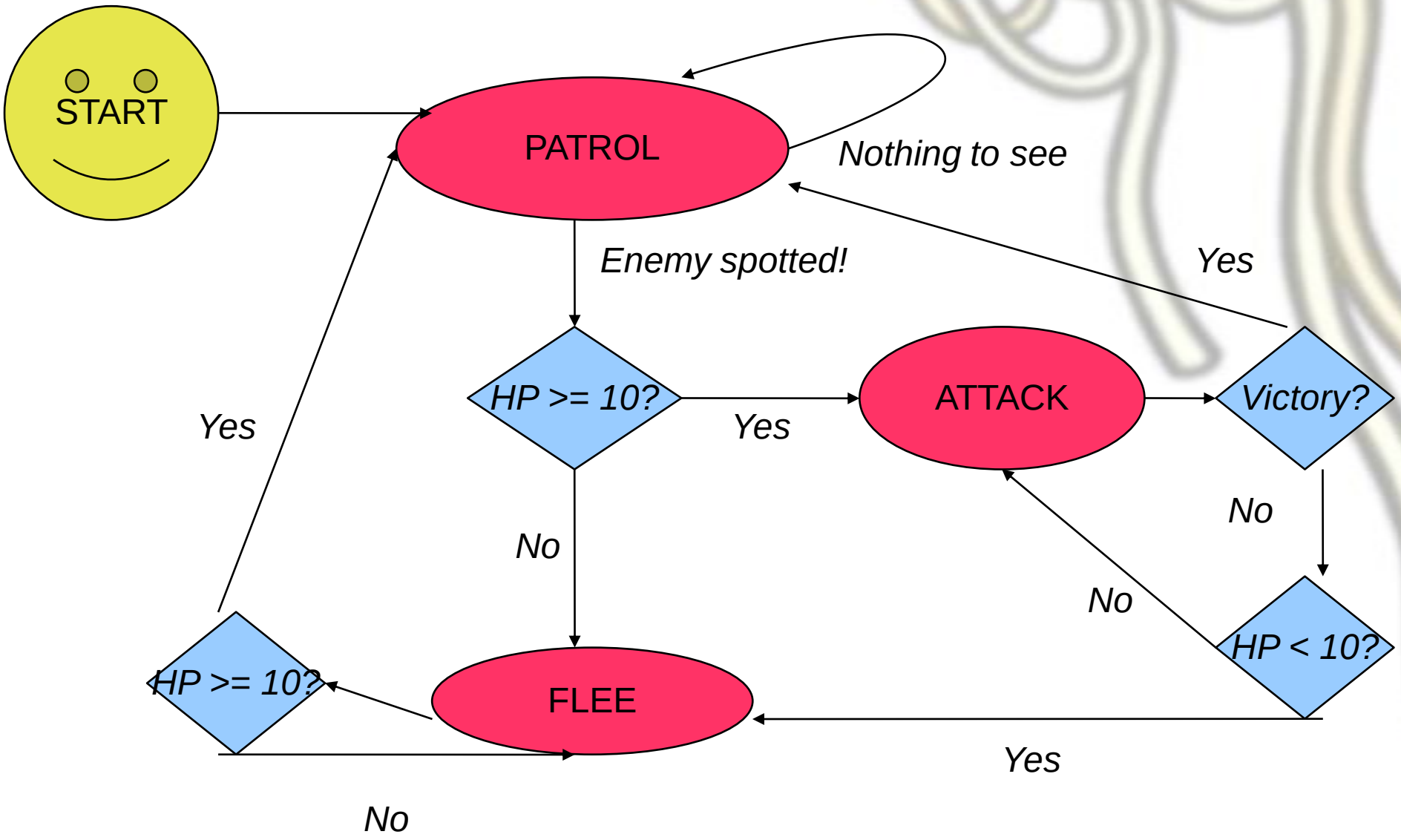
Further Uses for Coroutines

- Fibers provide a low-level mechanism for implementing coroutine like behavior
- Also useful for simplifying state machines:
  - Network Servers
  - Game Logic
  - Operating Systems

Game AI Example

- Games are concurrent simulations
- Typically event driven
- FSM style code
  - Difficult to write, debug and maintain
- Game logic eats up >90% of game code
  - [based on Quake 2, Arianne RPG, experience]
- Who knows how much development time?

A Simple AI State Machine



State Machine

```
switch(state)
{       case PATROL:
                foreach(n ; room.actors)
                {       if(n.is_hostile(this))
                        {       enemy = n;
                                if(hp > 10)
                                        state = ATTACK;
                                else
                                        state = FLEE;
                                break;
                        }
                }
        break;

        case ATTACK:
                if(!n.alive)
                {       state = PATROL;
                        break;
                }
                else if(hp >= 10)
                {       attack(enemy);
                        break;
                }
                else
                        state = FLEE;

        case FLEE:
                if(hp >= 10)
                        state = PATROL;
                else
                        flee();
        break;

        default: assert(false);
}
```

- Difficult control flow
  - Spaghetti code
- What about later modifications?
  - Suppose you need to put more states in?

Coroutine Version

```
while(true)
{
    foreach(n ; room.actors)
   {
            if(n.is_hostile(this))
            {
                while(n.alive && hp >= 10)
                        attack(n);

                while(hp < 10)
                        flee();

                break;
            }
    }□
    yield;
}
```

- Same result!
- State variables are implicit
  - No enemy, state etc.
- Easy to add more states

Game AI Conclusion

- Coroutines eliminate software engineering problems
- Make agent behavior part of language
  - States + Switch Statements => Global vars / GOTO
  - Coroutines => Structured Programming
- Why haven't we been using this all along?!

D MUD: A Case Study

- Build a MUD using Fibers for controlling agents
  - Developed by Matt Watkins (fellow MTU alum)
- Written entirely in D
  - 3k LoC
  - Full network support
  - GtkD GUI
- Developed in 1 week
- Very Flexible AI

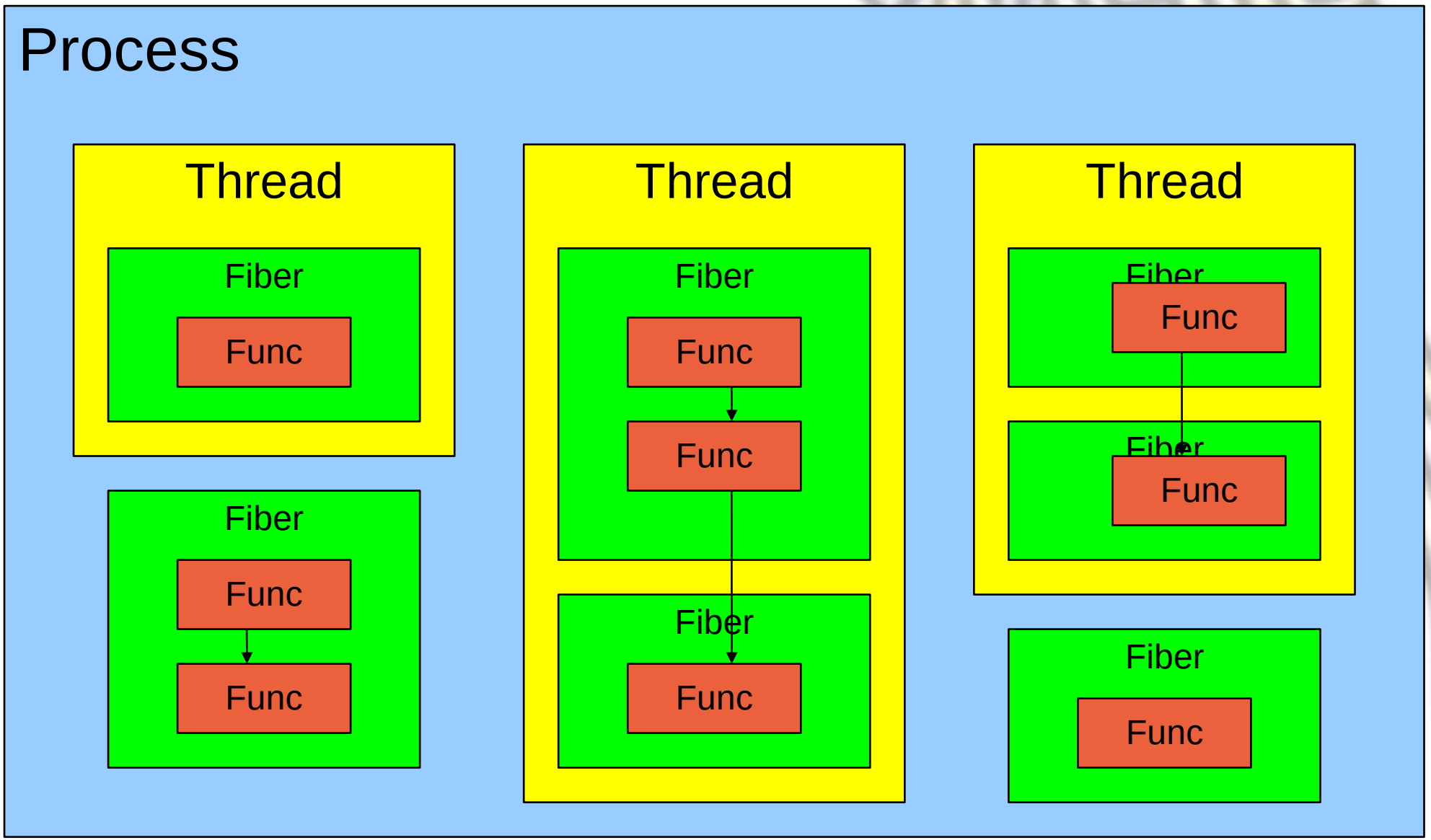
How to support coroutines?

- Can coroutines be implemented in D without modifying the language?
  - Let's not bother Walter if we don't have to
- If so, what are the minimal features we need?
  - Need to define interface
  - Implementation
  - Where they fit in the language/library environment

The Need for Fibers in Tango

- Runtime has to know about Fibers
  - Garbage collection
  - Threads
  - Exceptions
- Fiber implementation needs to know about system specific data structures
- Conclusion: Must implement in core library!

The Big Picture



The Fiber Interface

- Minimalist interface: 3 main functions

```
class Fiber
{
    //Creates a Fiber
    this(void delegate() func);

    //Calls a Fiber
    void call();

    //Yields active Fiber
    static void yield();
}
```

Implementation Requirements

- Fast
  - Switching contexts is going to happen a lot
- Low Overhead
  - Need to easily support thousands of processes
- Compatible with existing C/C++/D code
  - Must have a stack
- As portable as possible

Bad Idea #1: Threads

- This actually works and has been done:
  - See [Welch 2002] for a Java implementation
- Slow context switches
- Scalability Problems
  - Uses up scarce system resources (max 400 thread)
- Requires complex locking / synchronization

Bad Idea #2: Win32 Fibers

- Win2k/XP API for microthreads [Shankar2003]
- Not much faster than regular Threads
- Awkward Interface
  - Eats control of main thread, breaks GC, etc.
- Windows Only
- Officially Deprecated

The sigalt() Stack Trick

- Clever Unix hack [Engelschall 2000]
  - Uses sigalt to replace stack
  - Works in any C compiler
- In D, must port/reverse low-level syscalls & structures structures
  - (jmbuf_t, signal, etc.)
- **Does not work on Windows!!!!!**

Inline Assembler

- High Performance
  - Only 10 instructions per context switch
- C calling convention is standardized across x86
  - This includes Win32, Linux and Intel Macs!
- D inline assembler syntax is also standard
  - Works on both GDC and DMD
- ASM is most portable!

Some Final Thoughts

- Exceptions are tricky
  - Linux uses vectored exceptions, (easy)
  - Win32 uses SEH, must reverse engineer API (hard)
- Page faults/stack overflow
  - Can potentially implement dynamic stack
    - (in practice not yet…)
- Have to port for each platform
  - But then we'd have to do the same for sigalt/ucontext

Conclusion

- Enable concurrent programming
- Necessary part of the Tango runtime
  - Major advantage over Phobos
- Tango implementation is close to optimal
  - Without direct language support
- Can be implemented efficiently within

Future Directions

- More advanced models of concurrency
  - Actors
  - Communicating Sequential Processes
  - Pi-Calculus
- Deeper language integration
  - Can theoretically improve performance with context aware register allocation
- More awareness in programming communities

References

- Landin P. (1965) *A Generalization of Jumps and Labels.* UNIVAC Systems Programming Research
- Knuth D.E. (1968) *The Art of Computer Programming Vol. 1.* Addison-Wesley
- Liskov B. (1976) *Introduction to CLU.* MIT Press
- Gamma E., Helm R., Johnson R., Vlissides J. (1994) *Design Patterns.* Addison-Wesley
- Welch P. (2002) *CSP Networking for Java.* ICCS 2002
- Shankar A. (2003) *Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API.* MSDN.net
- Engelschall R.S. (2000) *Portable Multithreading.* USENIX 2000
- *Stackless Python* (2008) www.stackless.com

Acknowledgments

- Lars Ivar Igesund
- Sean Kelly
- Kris Bell
- Daniel Keep
- Matt Watkins
- CK Shene