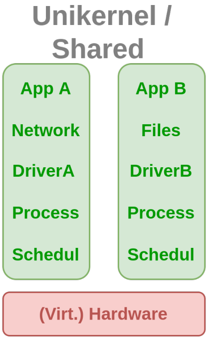
barkhauseninstitut.org

Compiling Unikernels into Micro Kernels

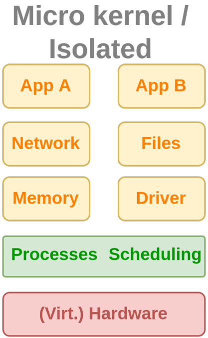
Diploma Defense

Lisza Zeidler

Performance vs. Security Trade-off



+ Performance
- No Isolation



- Overhead
+ Strong Isolation

Development and Verification

\+ Simple writing and testing
\+ Verifiable

\- Hard to develop and test
\- Verification hard/impossible

Wanted

Develop

shared memory,
single threaded,
no-isolation,
verified

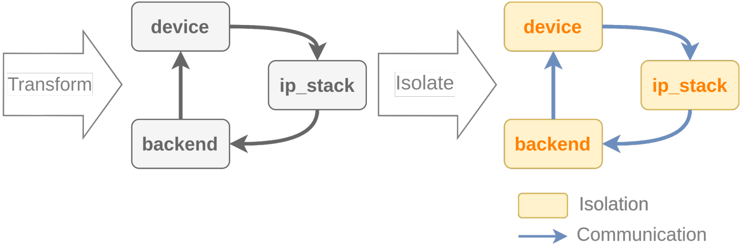
?

Deploy

isolated as
needed,
concurrent,
still verified

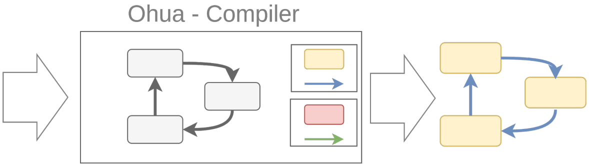
How to Generalizing the Rewrite ?

```
loop {
    device.get(...);
    ip_stack.unpack(...);
    backend.answer(...);
    ip_stack.pack(...);
    device.send(...);
}
```



Idea - Use a Compiler

```
loop {
   device.get(...);
   ip_stack.unpack(...);
   backend.answer(...);
   ip_stack.pack(...);
   device.send(...);
}
```



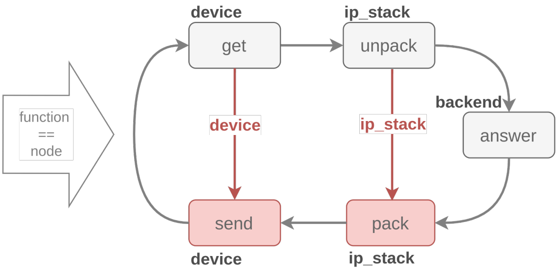
Ohua[1]:

- sequential $\rightarrow$ deterministic concurrent
- derives Data Flow Graph
- Backend Integrations provide process + channel implementations

[1] Sebastian Ertel, Christof Fetzer, and Pascal Felber. "Ohua: Implicit dataflow programming for concurrent systems". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. 2015, pp. 51–64.

Problem Solved? → No

```
loop {
    device.get(...);
    ip_stack.unpack(...);
    backend.answer(...);
    ip_stack.pack(...);
    device.send(...);
}
```



States are input and output of methods in data flow graphs

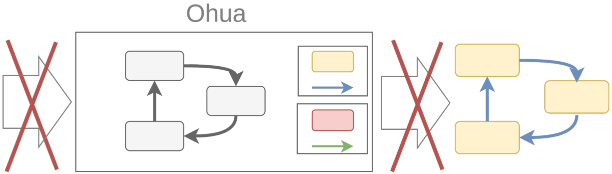
Requirement

State Locality: Isolated components or services should stay in their own runtime isolation

Task: Restructure the program, such that **every state** is used **exactly once**

Status Quo

```
loop {
    device.get(...);
    ip_stack.unpack(...);
    backend.answer(...);
    ip_stack.pack(...);
    device.send(...);
}
```

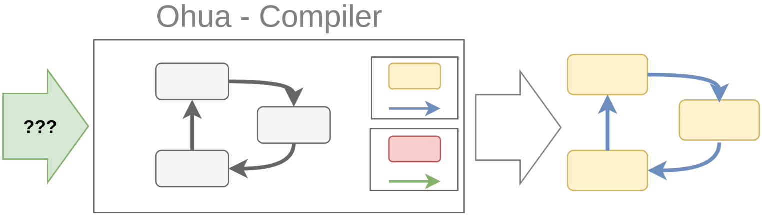


- program is no valid input
- output program would not meet requirements

Ohua does not support multiple state usage in loops or states in branches

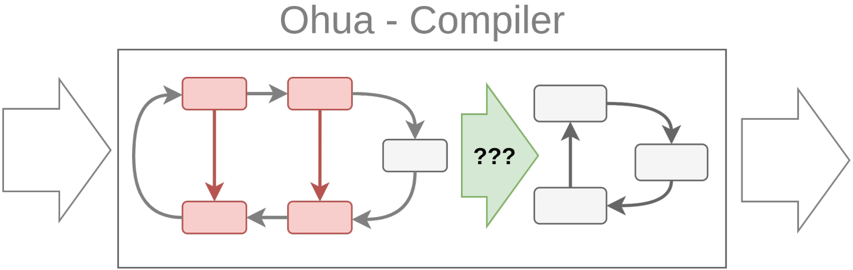
Question 1

```
loop {
    device.get(...);
    ip_stack.unpack(...);
    backend.answer(...);
    ip_stack.pack(...);
    device.send(...);
}
```



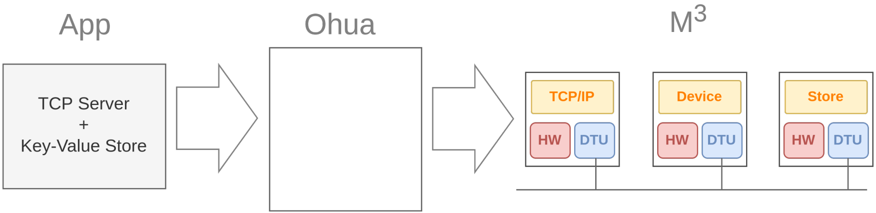
How to refactor to a valid input yielding state local programs?

Question 2



Can we teach those refactorings to Ohua?

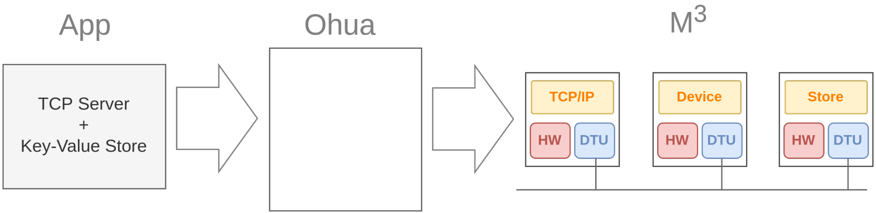
Concrete Example



- simple server app using `smoltcp`
- M^3 OS[2] as backend

[2] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. “M3x: Autonomous Accelerators via Context-Enabled Fast-Path Communication”. In: *USENIX Annual Technical Conference (ATC)*. Renton, WA, USA: USENIX, July 2019.

Concrete Example



- Goal: Run Device (NIC abstraction), TCP/IP-Stack and Key-Value Store as isolated processes in M^3

Concrete Example – App Structure

```
let store = Store::new();
/* intialization */
loop {
    ip_stack.poll(time, &mut device, & mut sockets);

    if let Some(input) = socket.recv(){
        if socket.can_send() {
            let outbytes = store.answer(&input);
            socket.send_slice(&outbytes[..]);
        }
    }
    phy_wait(dev_pointer, ip_stack.delay(time, &sockets));
}
```

Approach

Refactor a concrete application asking **"What would a compiler do?"**

Insight – Three Kinds of Refactorings

The Good : Solvable, formal Transformations

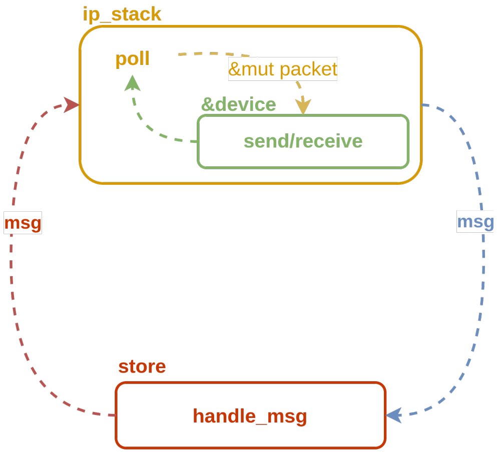
The Bad: Probably not-solvable, can be included in the Programming Model

The Ugly: Not-solvable, break the Promise

The Good

Refactoring Control Flow – Making States composable

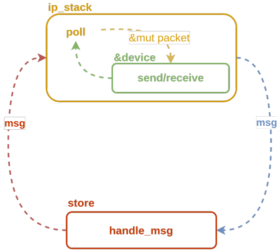
The Good - Restructure Sending Loop



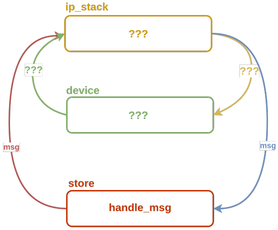
Situation

- outer loop: exchanging messages between `store` and `ip_stack`
- inner loop: exchanging packets between `ip_stack` and `device`

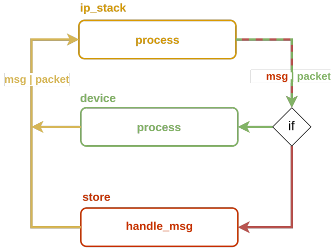
Plan



start →



lift device usage into scope →



merge data flow

Inner Loop

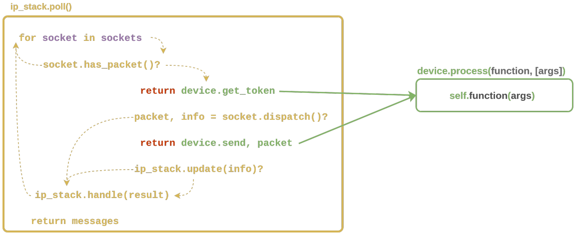
ip_stack.poll(&device)

```
for socket in sockets
  socket.has_packet()?
          device.get_token()?
          packet, info = socket.dispatch()?
          device.send(packet)?
          ip_stack.update(info)?
    ip_stack.handle(result)
  return messages
```

We need to call `device` outside of `ip_stack.poll()`

→ return calls to main scope

Inner Loop



- type of `ip_stack.poll()` and `device.process()` ?
- we can not ’send’ functions

Transformation 1 –Defunctionalization[3]

```
device.get_token
device.send(packet)
```

⇨

```
enum DeviceCall {
    GetToken,
    Send(IPPacket)
}
```

```
device.process(function, [args]) {
    self.function(args)
}
```

⇨

```
device.process(call:DeviceCall)-> ???
{
    match call {
       GetToken => self.get_toke(),
       Send(packet) => self.send(packet),
    }
}
```

1. define a sum type that represents functions and their arguments
   `enum DeviceCall`
2. define a function that *interprets* given values of that type back to function execution

[3] John C Reynolds. "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM annual conference-Volume 2*. 1972, pp. 717–740.

How to send step wise?

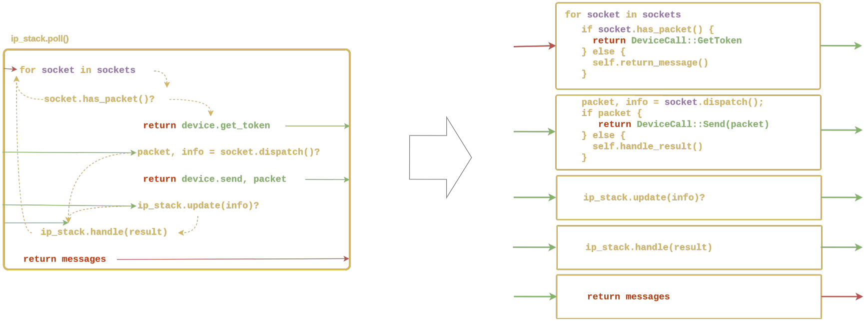
```
for socket in sockets
    socket.has_packet()?
                    return device.get_token
                   packet, info = socket.dispatch()?
                    return device.send, packet
                   ip_stack.update(info)?
   ip_stack.handle(result)
 return messages
```

We need to return to where we left

→ How to call sub-methods?

→ How to preserve state of execution?

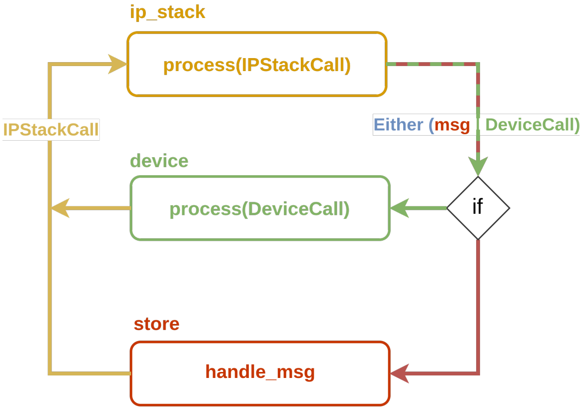
Transformation 2 – λ-Lifting[4]



turn closures ($\approx$ basic blocks) into top-level functions/ methods

[4] Thomas Johnsson. *Lambda lifting: Transforming programs to recursive equations*. Springer, 1985.

Defunctionalization – Closing the Loop



1. define a sum type that represents lifted functions `enum IPStackCall`
2. define `ip_stack.process` method to interpret the `IPStackCalls`
3. merge data flows

The Bad

Refactoring Reference Usage (in general)

The Bad – Trivial Case

References must not be used as function arguments in scope

Consequence: Programming model assumes for any `function(a, b, c)` in scope `a, b, c` are passed by value

The Bad – Non-trivial Example

Sending works via `tokens`:

1. `ip_stack` requests a sending `token` from `device`
2. `token` $\rightarrow$ reference to the device
3. `token.consume(inner_stack, |buffer| /*closure*/)` is called where `/*closure*/` writes the packet to the `buffer` provided by the `device`

$\Rightarrow$ Highly efficient in shared memory setting but ...

The Bad – Non-trivial Example

Sending works via `tokens`: $\Rightarrow$ Highly efficient in shared memory setting but

- How to split */*closure*/* to components?
- What to replace the `token` with?
- Are `tokens` useful without references?

Consequence: Refactoring involves dynamic/domain information the compiler does not have

The Ugly

Refactoring System Calls

The Ugly – System/OS calls

```
phy_wait(dev_pointer, ip_stack.delay(...));
```

wait for **File Pointer** until **Timeout**

```
maybe_syscall(a, b, c)
```

- System call?
- Supported by target OS?
- Can we replace by equivalent calls?

The Ugly – System/OS calls

- We can not identify system calls by syntax + static information
- Common practice→ provide annotations[5],[6],[7]
- What if different code structure is required?

Consequence:

Annotations are an options for simple cases

Complex cases break the Idea

[5] Hugo Lefeuvre. "FlexOS: easy specialization of OS safety properties". In: *Proceedings of the 22nd International Middleware Conference: Doctoral Symposium*. 2021, pp. 29–32.

[6] Vasily A Sartakov, Lluis Vilanova, and Peter Pietzuch. "CubicleOS: a library OS with software componentisation for practical isolation". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 546–558.

[7] Carolina Perez Ortega. "FlexC: Flexible Compartmentalization Through Automatic Policy Generation". PhD thesis. Massachusetts Institute of Technology, 2022.

Summary

Question 1 – Can we compile now?

Yes and No:

- add missing syntax support to Ohua
- adapt to M^3 supported types and `device` implementation

Summary

Question 2 – What could Ohua learn?

+ Defunctionalization can be used make states composable
+ λ-Lifting could be used to disentangle states
- We cannot refactor reference usage
- We cannot identify system calls

Summary

Question 2.5 – What's the cost?

Rewrite requires:

- owned data types
- copying data for sending
- multiple function calls instead of one in `ip_stack.poll`

$\Rightarrow$ Costs for rewrite on the same OS : TCP Packet throughput (Gb/s) decreased by $\approx$1/3

Conclusion

Tasks for us:

- Extend Ohuas State Support
- Decide what to compile and what to reject
- Derive Transformations where possible

Tasks of the programmer:

- Identify components and make every use explicit
- Stick to the programming model
- Know the target OS’s calls and types

And when it's all defined and written ...

The Bad – λ-Lifting – Problem

```
for socket in sockets
   if socket.has_packet() {
     return DeviceCall::GetToken
   } else {
     self.return_message()
   }
```

```
   packet, info = socket.dispatch();
   if packet {
      return DeviceCall::Send(packet)
   } else {
     self.handle_result()
   }
```

- We split a function stack
- Variables need to get from one Stack to the next

→ variables need to be heap allocated or send along with control flow

The Bad – λ-Lifting – Problem

```
for socket in sockets
   if socket.has_packet() {
     return DeviceCall::GetToken
   } else {
     self.return_message()
   }
```

```
   packet, info = socket.dispatch();
   if packet {
      return DeviceCall::Send(packet)
   } else {
     self.handle_result()
   }
```

Options

1. Send

→ not wanted for state internals

→ requires 'sendability'

2. Store in State

→ make the socket part of the `ip_stack` state

→ heap allocate

⇒ No internal cross referencing

The Bad – λ-Lifting – Problem

Splitting function stacks $\Rightarrow$ requirements for reference handling change

Consequence: **If** this transformation is applied, we need to extend the programming model to 'state internal' code $\rightarrow$ more complex requirements for the programmer

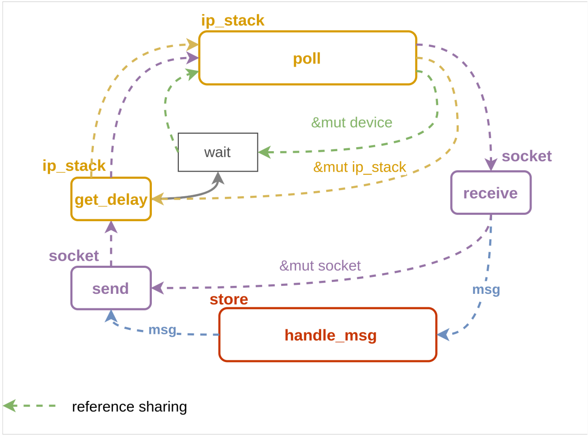
State Threads

```
let b = f(a)
```


```
let b = obj.handle(a)
```


```
let b = obj.handle(a);
let c = fun(b);
let final = obj.process(c);
```

Server Loop: Structure



Problems:

- `device` not used in scope
- `ip_stack` used more than once
- communication via shared references
- `socket` appear as component