EMULATING THE NINTENDO 3DS

Generative & Declarative Programming in Action

Tony Wasserka
@fail_cluez

CppCon 2018
24 September 2018

WHO AM I?

- Freelancer in embedded systems development
- Focus: Low-level & Type-safety
- Side projects: Game console emulators

PPSSPP Dolphin Citra

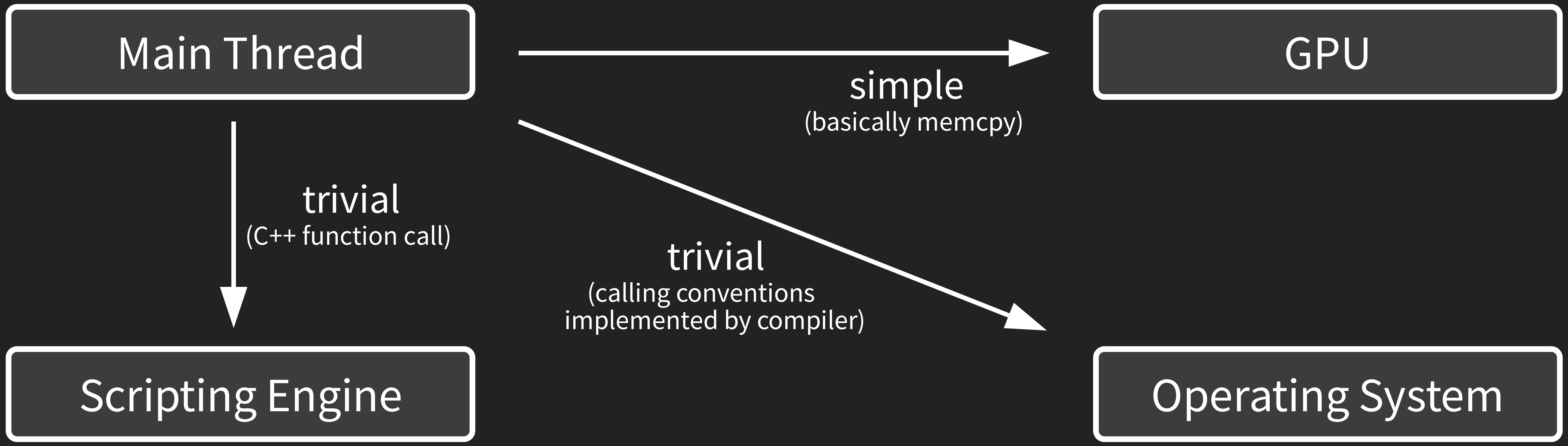
- Twitter: @fail_cluez
- GitHub: neobrain
- neobrain.github.io

WHAT IS THIS ABOUT?

- Serialization & emulation
- Case study: InterProcess Communication (IPC)
  - Generative & declarative
- How does modern C++ help?
  - How much boilerplate can we automate?
  - What runtime-errors can we turn into compile-errors?
  - How can we maximise reuse?

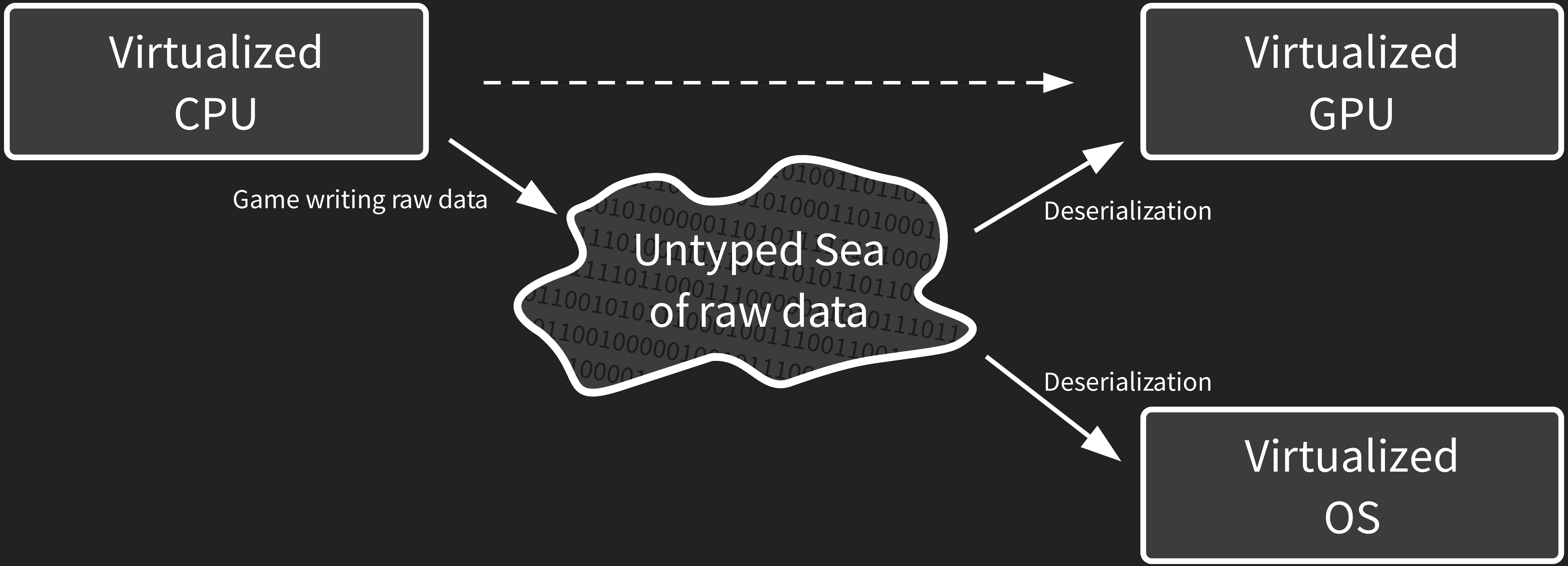
EMULATION & SERIALIZATION

Component interaction in games:



EMULATION & SERIALIZATION

Emulated Games:



EMULATION & SERIALIZATION

Example: System Call Emulation on ARM32

Virtual CPU

Register	Value
r0	0x1800600
r1	5
r2	0x1ff02000
r3	12
r4	0x200
…	…

svc 0x55 ⟶

Virtual Operating System

```
auto [result,dma] = SvcStartDma(LookupHandle(5),
                                0x1ff02000,
                                LookupHandle(12),
                                0x1800600,
                                0x200)
```

Presented at C++::London:
Generative Programming in Action: Emulating the 3DS

SERIALIZATION & EMULATION

A ubiquitous problem:

- System Calls: CPU registers ⟶ C++ function
- IPC: Memory ⟶ C++ function
- Emulated file IO: Disk ⟶ C++ struct
- GPU command buffers

What makes for reliable emulation?

- Avoid repetitive boilerplate
- Validate inputs (consistently!)
- Detect invalid states in the emulated system

Today's goal: Let the compiler deal with it!

THE NINTENDO 3DS

THE NINTENDO 3DS

- Released in 2011
- 2 CPU cores: ARMv6 @ 268 MHz
- Unique-ish GPU (DMP PICA200)
- 128 MB FCRAM
- Software stack:
  - Microkernel (fully multitasking)
  - About 40 active processes (microservices)
  - Games

THE 3DS SOFTWARE STACK

Game/Browser	Runs on emulated CPU
Processes ("Services")	API emulation (or could run on emulated CPU)
Kernel: Horizon	API emulation
ARM11 CPUs	Interpreter

PROCESS ARCHITECTURE

Functionality provided by external processes:

- Rendering graphics (gsp) & playing audio (dsp)
- Accessing WiFi (soc) & connecting to friends (frd)
- Loading assets & saving progress (fs)
- ...

~40 processes ("services") in total,
each serving different functionality

INTERPROCESS COMMUNICATION

Required to do anything useful on the 3DS!

- Client-Server based: Games ⇄ Services
- Request-response exchange via command blocks
- Marshalling of sensitive data by the OS kernel

Hierarchical: Game ⇄ cfg ⇄ fs ⇄ fspxi

IPC VISUALIZED

App: ReadFile

0: 0x802'02'05 (header)
1: 0x5
2: 0x200
3: 0x0
4: 0x100
5: 0x0
6: 0x200c
7: 0x1ff00200

→

Kernel

0: 0x802'02'05 (header)
1: 0x5
2: 0x200
3: 0x0
4: 0x100
5: 0x0
6: 0x200c
7: 0x2a700200

→

Service

0: 0x802'02'05 (header)
1: 0x5
2: 0x200
3: 0x0
4: 0x100
5: 0x0
6: 0x200c
7: 0x2a700200

→ Emulation ⚙

IPC VISUALIZED

App		Kernel		Service: Response
0: 0x802'00'04 (header)	⟵	0: 0x802'00'04 (header)	⟵	0: 0x802'00'04 (header)
1: 0x0		1: 0x0		1: 0x0
2: 0x100		2: 0x100		2: 0x100
3: 0x0		3: 0x0		3: 0x0

EMULATING IPC COMMAND HANDLERS

SVPIR: Common dispatch flow:

- Select C++ handler function based on command index

```
std::tuple<Result,uint32_t> DoReadFile(uint32_t, uint64_t, uint64_t, BufferPointerW)
```

- Verify command header (number of parameters)

```
(cmd_header & 0xFF == 5)  &&  ((cmd_header >> 8) & 0xff == 2)
```

- Parse parameters from command block

header 0x8020205	uint32 5	uint64_lo 0xdeadbeef	uint64_hi 0x5555	uint64_lo 0xd00f	uint64_hi 0	buffer descriptor …	buffer addr 0x1ff00200

- Invoke C++ handler function

```
DoReadFile(5, 0x5555deadbeef, 0xd00f, BufferPointerW{0x1ff00200});
```

- Write Response back to command block

header: 0x8020002	Result: 0x0	Result 2: 0xd00f

EMULATING IPC COMMAND HANDLERS

How often do we need to write the SVPIR logic?

- ~40 active processes
- Each with ~30 IPC commands on average
- Manual glue to invoke the C++ handler required for each

That is a lot of work.

Correctness? Consistency? Maintainability?

Enter declarative & generative programming

DECLARATIVE PROGRAMMING

- Let's take a step back
- Focus on the problem description first
- Find a solution later, and make it generic

Separate the **what** from the **how**

IPC COMMANDS

How to characterize IPC commands? At compile-time

- Command id
- Request data: List of "normal" parameters
- Request data: List of "special" parameters
- Response data: Another 2 lists

E.g. FS::OpenFile:

- Command id 0x802
- Request data: `IOFlags`, `FileAttributes`, `uint32_t`
- Request data: `StaticBuffer`
- Response data: `FileDescriptor` (no "special"s)

A DECLARATIVE INTERFACE

Using types for information storage:

```
namespace FS { // FileSystem-related commands

using OpenFile    = IPCCmd<0x802>
                    ::normal<IOFlags, FileAttributes, uint32_t>
                    ::special<StaticBuffer>
                    ::response<FileDescriptor>;

using GetFileSize = IPCCmd<0x804>
                    ::normal<FileDescriptor>
                    ::special<>
                    ::response<uint64_t>;

}
```

⇒ Builder-like pattern

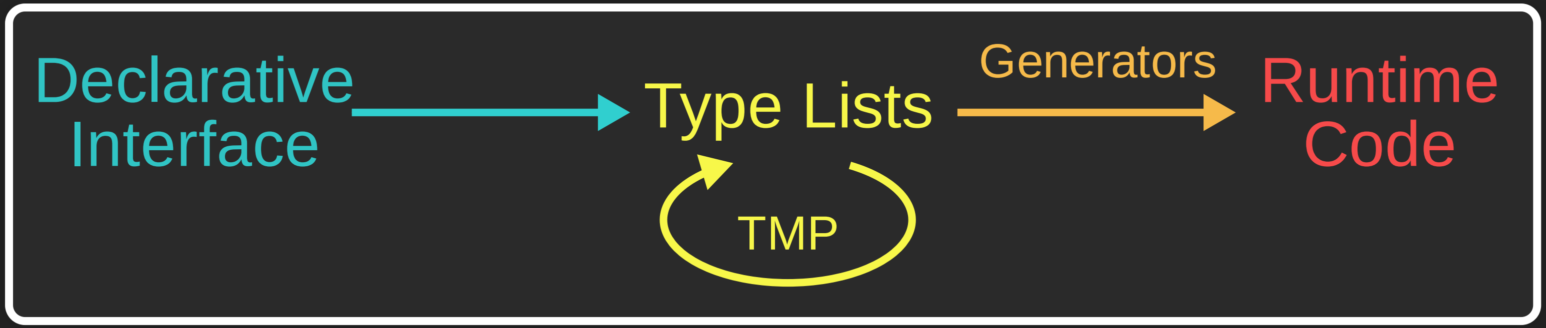
A DECLARATIVE INTERFACE

```
template<uint32_t CommandId>
struct IPCCmd {
  template<typename... NormalParams>
  struct normal {
    template<typename... SpecialParams>
    struct special {
      // Export template params in the interface
      static constexpr uint32_t command_id = CommandId;
      using normal_params    = std::tuple<NormalParams...>;
      using special_params   = std::tuple<SpecialParams...>;
    };
  };
};

namespace FS { // FileSystem-related commands
using OpenFile    = IPCCmd<0x802>
                  ::normal<IOFlags, FileAttributes, uint32_t>
                  ::special<StaticBuffer>;
using GetFileSize = IPCCmd<0x804>
                  ::normal<FileDescriptor>
                  ::special<>;
}
```

DECLARATIVE COMPILE-TIME PROGRAMMING

Building blocks:



Kinds of Declarative Interfaces:

- Type-based systems
- constexpr objects
- Reflection-based systems
- Plain definition vs eDSL

OUR VISION FOR SVPIR

C++ Handler

```
std::tuple<Result, uint32_t>
DoReadFile(FileDesc fd,
           uint64_t offset,
           uint64_t num_bytes,
           WriteableBuffer& output)
```

Declarative Interface

```
using ReadFile =
    IPCCmd<0x803>
    ::normal<FileDesc, uint64_t, uint64_t>
    ::special<WriteableBuffer>
    ::response<uint32_t>;
```

↓ Extract std::tuples ↓

```
using RequestList = ReadFile::request_list;
using ResponseList = ReadFile::response_list;
```

+ Generators

```
template<typename Cmd, typename... T> tuple<T...> DecodeMessage(CmdBlock&)
template<typename Cmd, typename... T> void         EncodeMessage(CmdBlock&, T... data)
```

↓ Combine ↓

```
GlueCommandHandler<ReadFile>(cmd_block, DoReadFile)
```

OUR VISION

Advantages:

- Automated command encoding/decoding
- Type safe: Decoded data matches C++ function signature *by design*
- Bonus: It's easier to read (since there's nothing to read anymore)

GENERATORS

Core idea: Generate runtime code based on a type list via

- Recursion
- `for_each(tuple, f)`
- parameter pack expansions (C++11)
- fold expressions (C++17)

We got our type list from the declarative interface

How do we generate a command block decoder?

A GENERATIVE DECODER

```
RequestList: std::tuple<uint32_t, uint64_t, uint64_t, WriteableBuffer >
```

header	uint32	uint64_lo	uint64_hi	uint64_lo	uint64_hi	buffer descriptor	buffer addr
0x8020205	5	0xdeadbeef	0x5555	0xd00f	0	…	0x1ff00200

```
// Read a single entry from the CmdBlock and advance "offset"
template<typename T>
auto DecodeEntry(int& offset, CmdBlock& block) { ... }

// Iterate over entire CmdBlock & gather results & apply to "handler"
template<typename Handler, typename... Ts>
auto DecodeAllAndApply(CmdBlock& cmd_block, Handler&& handler) {
  int offset = 0x1; // into command block
  return handler(DecodeEntry<Ts>(offset, cmd_block)...);
}
```

GENERATORS

```
RequestList: std::tuple<uint32_t, uint64_t, uint64_t, WriteableBuffer >
```

header	uint32	uint64_lo	uint64_hi	uint64_lo	uint64_hi	buffer descriptor	buffer addr
0x8020205	5	0xdeadbeef	0x5555	0xd00f	0	…	0x1ff00200

Decoding 32-bit values:

```
template<typename T>
auto DecodeEntry(int& offset, CmdBlock& block) {
  if constexpr (std::is_same_v<T, uint32_t>) {
    return block.ReadU32(offset++);








  } else {
    ...
  }
}
```

GENERATORS

```
RequestList: std::tuple<uint32_t, uint64_t, uint64_t, WriteableBuffer >
```

header 0x8020205	uint32 5	uint64_lo 0xdeadbeef	uint64_hi 0x5555	uint64_lo 0xd00f	uint64_hi 0	buffer descriptor …	buffer addr 0x1ff00200

Decoding 64-bit values:

```
template<typename T>
auto DecodeEntry(int& offset, CmdBlock& block) {
  if constexpr (std::is_same_v<T, uint32_t>) {
    return block.ReadU32(offset++);
  } else if constexpr (std::is_same_v<T, uint64_t>) {
    uint32_t val_low  = block.ReadU32(offset++);
    uint32_t val_high = block.ReadU32(offset++);
    return (val_high << 32) | val_low;



  } else {
    ...
  }
}
```

GENERATORS

`RequestList: std::tuple<uint32_t, uint64_t, uint64_t, WriteableBuffer >`

header	uint32	uint64_lo	uint64_hi	uint64_lo	uint64_hi	buffer descriptor	buffer addr
0x8020205	5	0xdeadbeef	0x5555	0xd00f	0	…	0x1ff00200

Decoding buffer descriptors:

```
template<typename T>
auto DecodeEntry(int& offset, CmdBlock& block) {
  if constexpr (std::is_same_v<T, uint32_t>) {
    return block.ReadU32(offset++);
  } else if constexpr (std::is_same_v<T, uint64_t>) {
    ...
  } else if constexpr (std::is_same_v<T, WriteableBuffer>) {
    uint32_t descriptor = block.ReadU32(offset++);
    auto [size, flags] = DecodeBufferDescriptor(descriptor);
    uint32_t address = block.ReadU32(offset++);
    return WriteableBuffer { address, size };
  } else {
    ...
  }
}
```

A GENERATIVE DECODER

```
RequestList: std::tuple<uint32_t, uint64_t, uint64_t, WriteableBuffer >
```

header	uint32	uint64_lo	uint64_hi	uint64_lo	uint64_hi	buffer descriptor	buffer addr
0x8020205	5	0xdeadbeef	0x5555	0xd00f	0	…	0x1ff00200

```
// Read a single entry from the CmdBlock and advance "offset"
template<typename T>
auto DecodeEntry(int& offset, CmdBlock& block) { ... }

// Iterate over entire CmdBlock & gather results & apply to "handler"
template<typename Handler, typename... Ts>
auto DecodeAllAndApply(CmdBlock& cmd_block, Handler&& handler) {
  int offset = 0x1; // into command block
  return handler(DecodeEntry<Ts>(offset, cmd_block)...);
}
```

No boilerplate!

Caveat 1: The template needs a std::tuple<T...>

GENERATORS:

```
RequestList: std::tuple<uint32_t, uint64_t, uint64_t, WriteableBuffer >
```

header	uint32	uint64_lo	uint64_hi	uint64_lo	uint64_hi	buffer descriptor	buffer addr
0x8020205	5	0xdeadbeef	0x5555	0xd00f	0	…	0x1ff00200

```
// Read a single entry from the CmdBlock and advance "offset"
template<typename T>
auto DecodeEntry(int& offset, CmdBlock& block) { ... }

template<typename TypeList> struct DecodeAllAndApply;

template<typename... Ts>
struct DecodeAllAndApply<std::tuple<Ts...>> {
  int offset = 1; // offset into command block

  // Iterate over entire CmdBlock & gather results & apply to "handler"
  template<typename Handler>
  auto operator()(CmdBlock& cmd_block, Handler&& handler) {
    // Caveat 2: Execution order undefined :(
    return handler(DecodeEntry<Ts>(offset, cmd_block)...);
  }
};
```

GENERATORS:

DEMO TIME!

01a_generators.cpp
01b_generators.cpp
magic.hpp

GENERATORS: RESULT ENCODER

header	Result	uint32_t
0x8020002	0x0	0xd00f

Trivial with fold expressions!

```
template<typename T>
void EncodeEntry(int& offset, CmdBlock& block, T t) { ... }

template<typename... Ts>
void EncodeAll(CmdBlock& cmd_block, Ts... ts) {
  int offset = 1;

  (EncodeEntry<T>(offset, cmd_block, ts), ...);
}
```

GENERATORS WITH DECLARATIVE INTERFACES

```
template<typename IPCRequest, typename Handler>
void GlueCommandHandler(CmdBlock& cmd_block, Handler&& handler) { // S
  auto request_header = cmd_block.ReadU32(0);
  if (request_header != IPCRequest::request_header) // V
    throw std::runtime_error("Invalid request header");

  auto results = DecodeAllAndApply<IPCRequest::request_list>{}(cmd_block, handler); // P + I
  cmd_block.WriteU32(IPCRequest::response_header);
  EncodeAll<IPCRequest::response_list>(cmd_block, results); // R
}
```

This can be used for all IPC commands!

DECLARATIVE INTERFACES: GENERATORS

Declarative approach maximizes reusability

⇒ Trivial bringup of entire subsystems!

```
using GetFileSize = IPCCmd<0x804>
                    ::normal<FileDescriptor>::special<>
                    ::response<uint64_t>;
```

↓

Handlers

GlueHandler<GetFileSize>(cmdblk, DoGetFileSize)

↓

DoGetFileSize(FileDesc{5})

↓

Synthesis

auto blk = CraftBlock<FS::GetFileSize>(fd)

↓

0: 0x804'00'01 (header)
1: 0x5
2: 0x0

↓

Logging

std::cout << LogInfo<FS::GetFileSize>

↓

"GetFileSize: fd 5"

DECLARATIVE INTERFACES & GENERATORS

DEMO TIME!

02_generators.cpp
dummy_env.hpp
ipc.hpp
magic.hpp

WHY DECLARATIVE?

- Separates concerns (what vs how)
  - Business structure and logic rarely *both* change
- Speeds up feature bringup due to reusable components
- Expresses programmer intent naturally
- Encourages automation of boilerplate generation
- Helps detect errors at compile-time

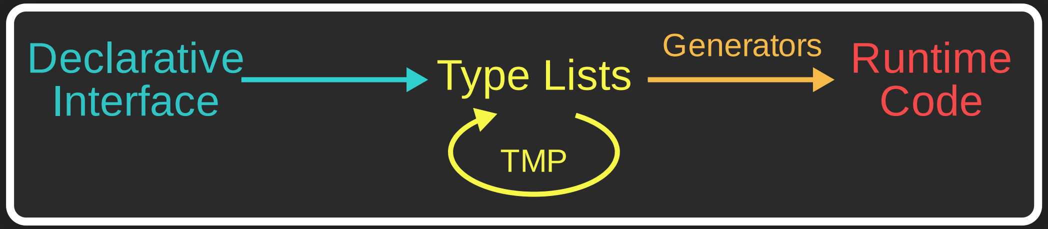
Improved flexibility, time to market, and maintainability!

CONCLUSION

- Untyped data makes serialization centric to emulation
- Generating code via stateful variadic folds over type lists

  Fold expressions are big for simplicity!

- Declarative interfaces: Novel but powerful



- Vastly more maintainable and expressive at zero overhead

THANKS!

neobrain.github.io

@fail_cluez

neobrain

(const west const best!)