BRINGING RENDERSCRIPT TO LLDB

Ewan Crawford / Luke Drummond

Codeplay Software

codeplay

THE HETEROGENEOUS SYSTEMS EXPERTS

- Heterogeneous systems experts
- Based in Edinburgh, Scotland

CONTENTS

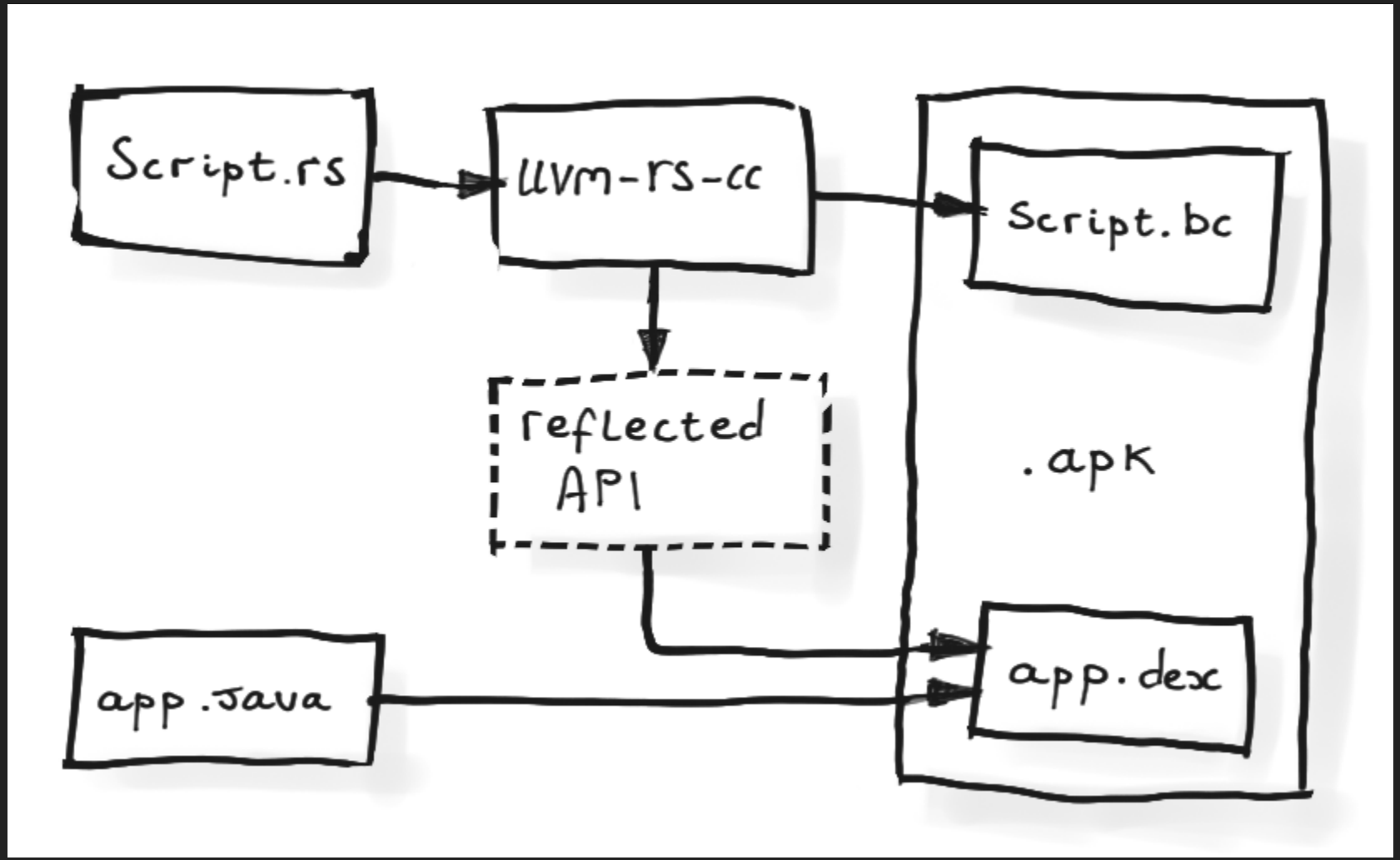
LEARN YOU A RENDERSCRIPT

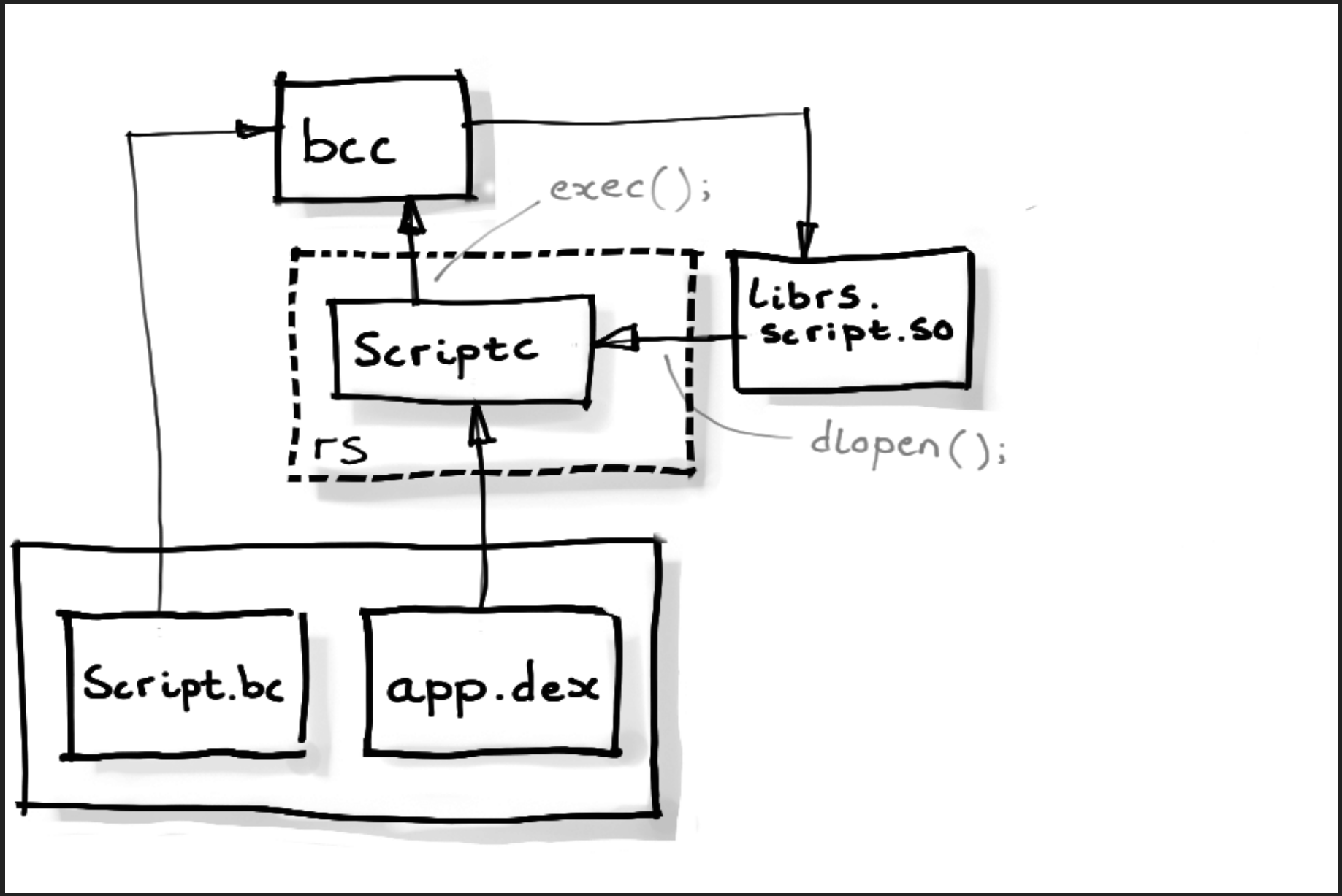
WHAT IS RENDERSCRIPT?

- RenderScript is Android's heterogeneous compute API
- Portable acceleration across wide range of Android devices
- Java host code dispatches C99-like kernels from Scripts

SLANG & BCC COMPILERS

- Slang frontend consumes Scripts containing Kernels and emits portable IR pushed to device
- `bcc` backend runs on device, performing some RS specific IR passes


Script.rs
llvm-rs-cc
Script.bc
reflected
API
.apk
app.java
app.dex


bcc
exec();
Scriptc
rs
libRS.
script.so
dlopen();
Script.bc
app.dex

ALLOCATIONS

- An allocation is a container for data which is passed to and from RenderScript scripts
- Data type can be simple e.g. `uchar4`, or a more complex C `struct`
- Organized in up to 3 dimensions: `{x, y, z}`

RENDERSCRIPT RUNTIME

RENDERSCRIPT LIBRARIES

- libRS.so
- libRSDriver.so
- libRSCpuRef.so
- Compiled scripts, e.g. librs.foo.so

BCC .EXPAND

```
void fookernel.expand(RsExpandKernelDriverInfo* p, ...)
```

Current thread coordinate:

- x: local variable called `rsIndex`
- y: `p->current.y` in `fookernel.expand`
- z: `p->current.z` in `fookernel.expand`

.EXPAND DEBUG INFO

- `.expand` IR has no debug info so how can we inspect thread coordinates?
- We generate DWARF with spoofed source file generated.rs and language DW_AT_GOOGLE_RenderScript

LLDB RUNTIME

PLUGIN ARCHITECTURE

LLDB has functionality modules which are loaded dynamically at runtime depending on environment

- PluginObjectFileELF
- PluginABISysV_hexagon
- PluginPlatformAndroid

RENDERSCRIPT LANGUAGE RUNTIME

Lives in Plugins/LanguageRuntime/RenderScript

```
(lldb) help language renderscript
The following subcommands are supported:

  allocation -- Commands that deal with renderscript allocations.
  context    -- Commands that deal with renderscript contexts.
  kernel     -- Commands that deal with renderscript kernels.
  module     -- Commands that deal with renderscript modules.
  status     -- Displays current renderscript runtime status.
```

KERNEL BREAKPOINT COMMAND

- Narrows search scope to RS Script modules
- Fall back to `.expand` if kernel name can't be found
- User can set a specific invocation to break on
- Extensibility for future accelerator targets

INSPECTING TARGET

```
// We override from LanguageRuntime
void
RenderScriptRuntime::ModulesDidLoad(const ModuleList &module_list)
```

- Our plugin constructor invokes `ModulesDidLoad()` with all the currently loaded modules
- Detects RS libraries and caches a local copy
- Triggers events such as placing hooks and breaking out of wait for attach loop

MANUAL SYMBOL PARSING

```
// Find symbol name, e.g. .rs.info from data section
const Symbol *info_sym = m_module->FindFirstSymbolWithNameAndType(
                                        ConstString(".rs.info"),
                                        eSymbolTypeData);

// Get file address of symbol
const addr_t addr = info_sym->GetAddressRef().GetFileAddress();
const addr_t size = info_sym->GetByteSize();

const FileSpec fs = m_module->GetFileSpec();
const DataBufferSP buffer = fs.ReadFileContents(addr, size);
```

HOOKS

WHAT IS A HOOK?

- Internal breakpoint on a function with a callback handler
- Optional baton holds persistent data we'd like to check in the callback

```
void
Breakpoint::SetCallback(BreakpointHitCallback callback,
                        void *baton, bool is_synchronous);
```

BREAK ON A KERNEL COORDINATE

- Software watchpoint
- Callback inspects `.expand` frame for thread coordinates
- Baton contains coordinate user has asked to break on
- Break back to user if thread variables match baton

```
std::array<std::string> var_names{
    "rsIndex", "p->current.y", "p->current.z"
};

for (auto &name : var_names) {
  auto val_sp =
    frame_sp->GetValueForVariableExpressionPath(name, ...);
}
```

HOOKING ALLOCATION CREATION

- Break on the mangled symbol because debug info isn't present
- Inspect parameters using register & stack reading code for target ABI

```
// From libRSDriver.so
rsdAllocationInit(
    const Context *rsc, Allocation *alloc, bool forceZero
);
```

```
_Z17rsdAllocationInitPKN7android12renderscript7ContextEPNS0_10Allocation
```

Gives us a pointer to internal representation of Allocation, but we can't infer anything more from that. So how do we proceed?

JIT THE ALLOCATION DETAILS!

- `lldb` is linked with LLVM and uses the `clang` frontend to JIT expressions
- JIT functions/data objects living in the runtime

JITTING THE RENDERSCRIPT RUNTIME

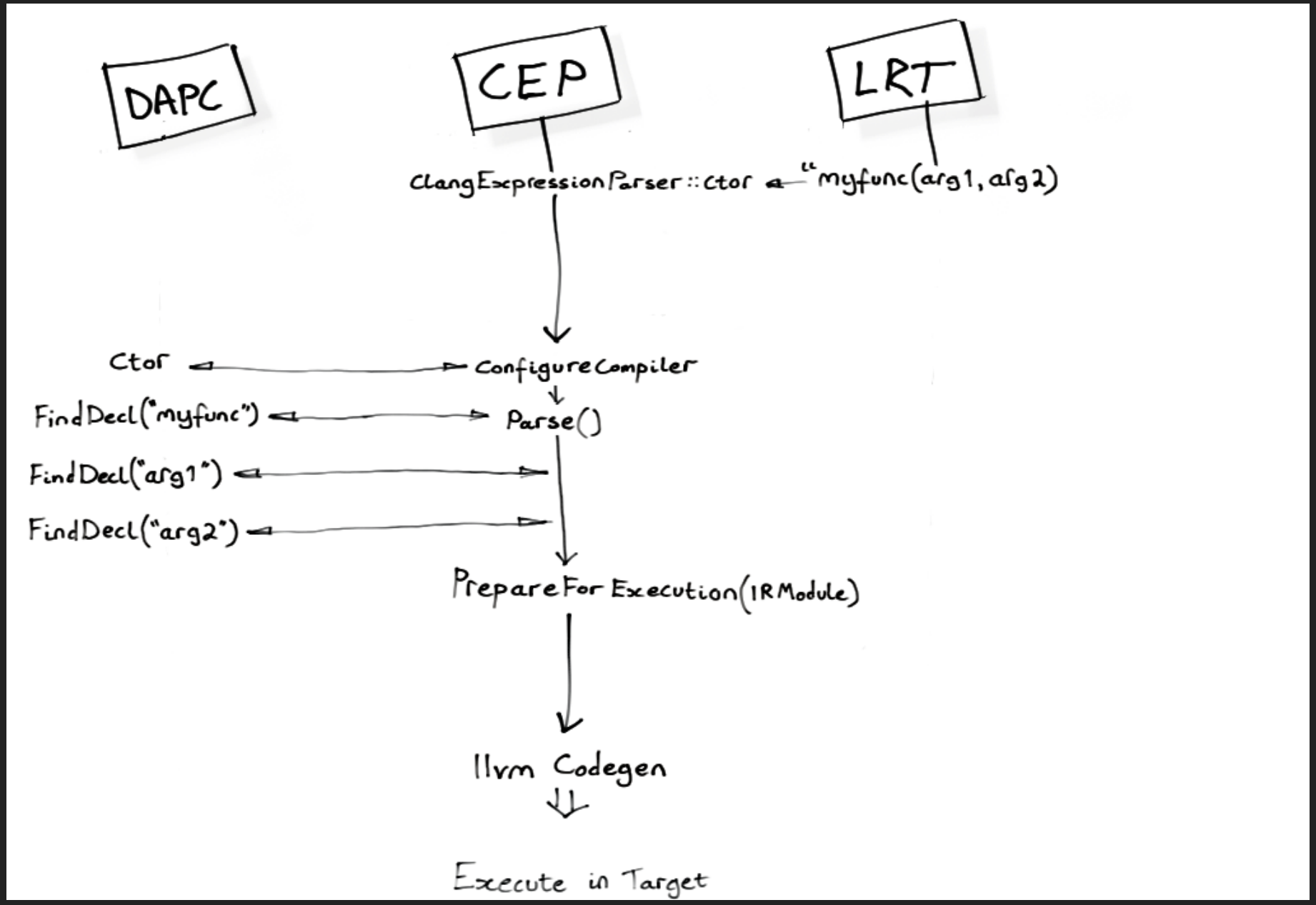
```
// In a descendant of `LanguageRuntime`

EvaluateExpressionOptions opts;
ValueObjectSP expr_result;
ExecutionContext exe_ctx;
GetProcess()->CalculateExecutionContext(exe_ctx);
opts.SetLanguage(lldb::eLanguageTypeC99);

GetProcess()->GetTarget()->EvaluateExpression(
    "add_two_ints(4, 5)",
    exe_ctx->GetFramePtr(),
    expr_result,
    opts
);
::printf("4 + 5 == %s", expr_result->GetValueAsCString());
```

INTERACTIVE KERNEL-SIDE JITTING

- RenderScript API functions live in `libclcore.bc` linked by `bcc` into the script on device
- This means llvm JIT ABI doesn't always line up with `bcc` sneaky compiler tricks


DAPC
CEP
LRT
ClangExpressionParser::Ctor
"myfunc(arg1, arg2)
Ctor
ConfigureCompiler
FindDecl("myfunc")
Parse()
FindDecl("arg1")
FindDecl("arg2")
PrepareForExecution(IRModule)
llvm Codegen
Execute in Target

ABI ISSUES

1. `llvm-rs-cc` generates ARM IR at the frontend
2. However x86 is register-poor
3. `SIGSEGV (0x00000bad)`

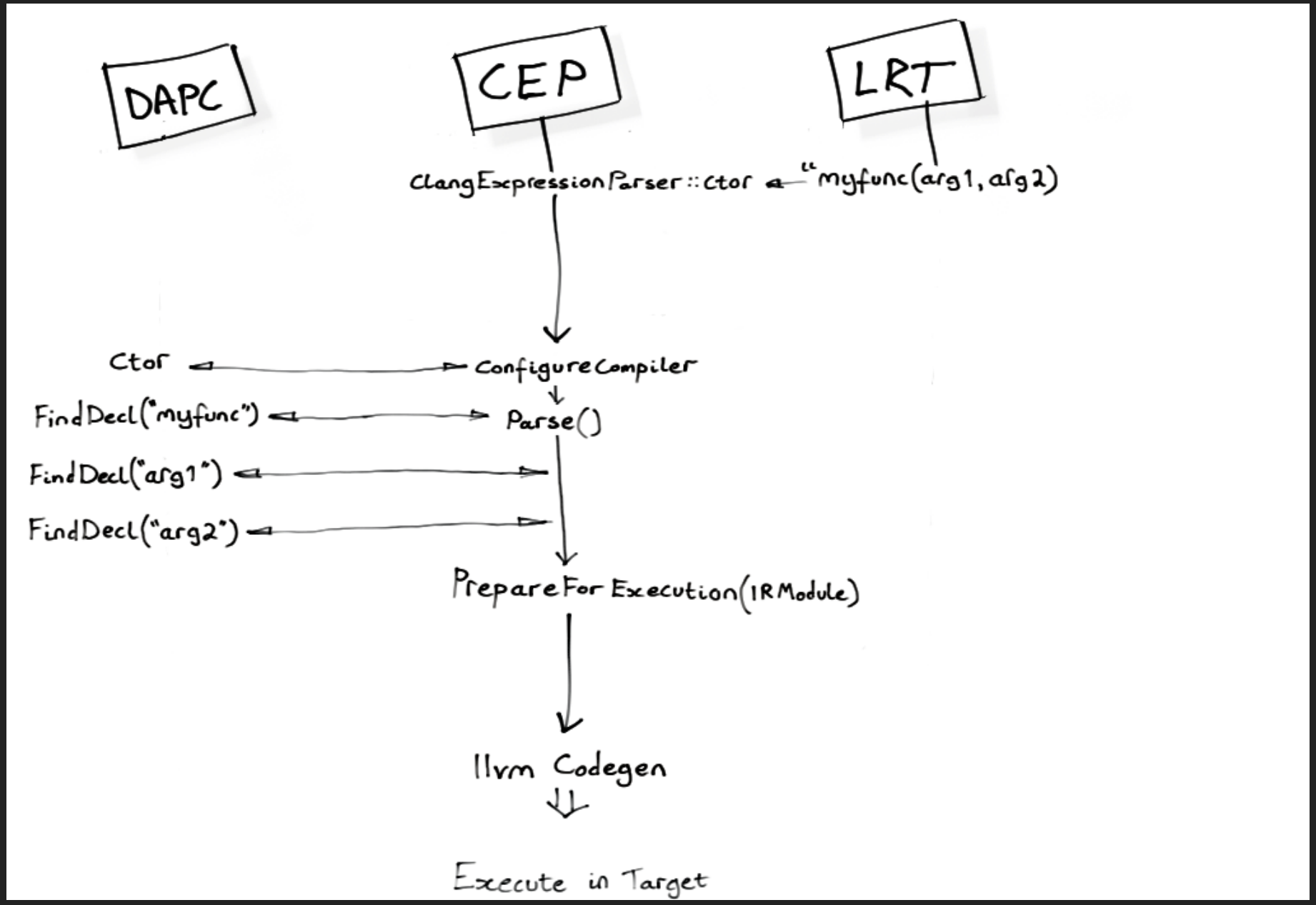
- `sizeof(rs_allocation) == 32;`
- Therefore `rs_allocation` is 256bits
- So `rs_allocation` is returned on the stack

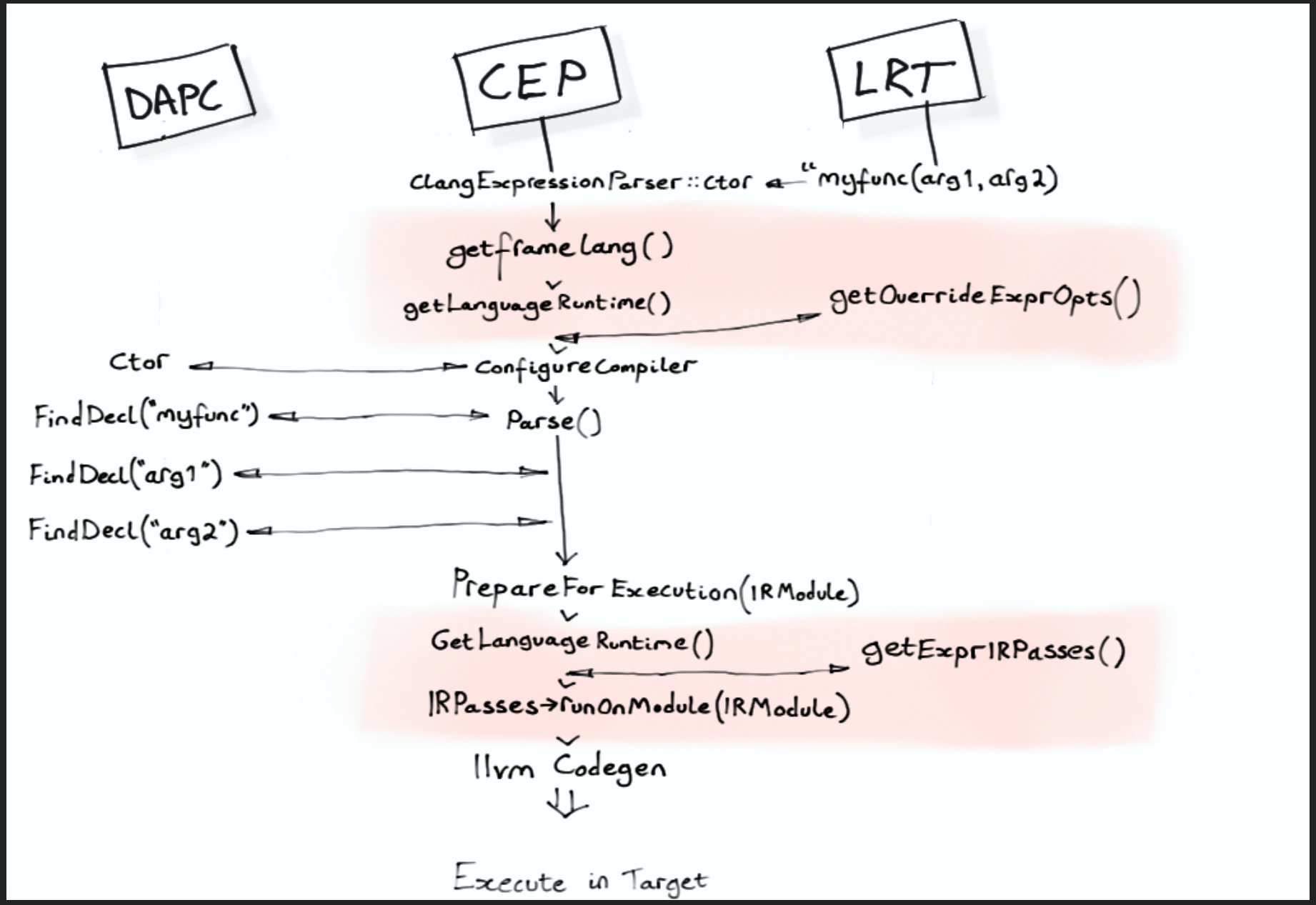
```
// Wrong! i686 can't return objects this large directly
long4 clamp(long4 val, long4 min, long4 max);
```

```
// This is what we should see
long4 *clamp(long4 *retval, long4 val, long4 min, long4 max);
```

Modifications we made to the JIT

1. Detect RenderScript from DW_AT_language for the stopped frame
2. API for querying LanguageRuntime plugin for `clang::TargetOptions`
3. A way to fixup the ABI for function calls
4. Run custom `llvm::ModulePass` from LanguageRuntime


DAPC
CEP
LRT
ClangExpressionParser::Ctor
"myfunc(arg1, arg2)
Ctor
ConfigureCompiler
FindDecl("myfunc")
Parse()
FindDecl("arg1")
FindDecl("arg2")
PrepareForExecution(IRModule)
llvm Codegen
Execute in Target


DAPC
CEP
LRT
ClangExpressionParser::Ctor
"myfunc(arg1, arg2)
getFrameLang()
getLanguageRuntime()
getOverrideExprOpts()
Ctor
ConfigureCompiler
FindDecl("myfunc")
Parse()
FindDecl("arg1")
FindDecl("arg2")
PrepareForExecution(IRModule)
GetLanguageRuntime()
getExprIRPasses()
IRPasses->runOnModule(IRModule)
llvm Codegen
Execute in Target

WHAT'S NEXT

- Debugging hardware accelerated RenderScript
- Script Groups
- Autoloading the runtime

ewan@codeplay.com

luke.drummond@codeplay.com

special thanks to Aidan Dodds for the diagrams