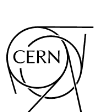


EP-SFT
Software Frameworks and Tools

PRINCETON
UNIVERSITY

CppInterOp: Interactive C++ as a Service and Advanced Language Interoperability

Aaron Jomy
for the Compiler-Research group, ROOT Team @CERN

`EuroLLVM 2026, Dublin`

Contents

COMPILER C|R RESEARCH

❑ Interactive C++

❑ As a Service (CppInterOp)

❑ Advanced Language Interoperability (CppInterOp clients)

❑ Demo

Motivation

COMPILER C|R RESEARCH

Can we combine the power of C++ with the expressiveness of Python?

- The C++ language remains an industry standard for production-grade, performance-critical system software.
- But for open problems: does not provide an easy path for rapid prototyping
  - Edit-recompile-debug cycles introduce latency
    Hard to interactively explore data/algorithms
  - Dominated by more data science friendly programming languages/frameworks, at the cost of performance

Motivation

COMPILER C|R RESEARCH

Exploratory programming

- Converge from prototype to production

Rapid Iteration

- Evolution of classes/data structures without recompilation
- Redefining entities

Interactivity

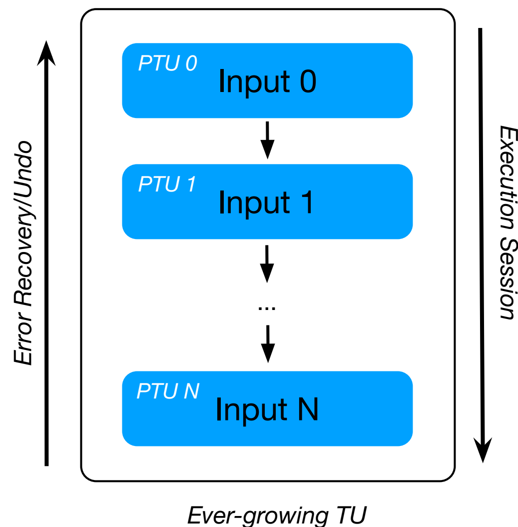
- Interactive data exploration
- Interactive usage of a C++ library, e.g. learning new frameworks

Interactive C++

Incremental compilation in Clang with clang::Interpreter

Incremental Compilation in Clang

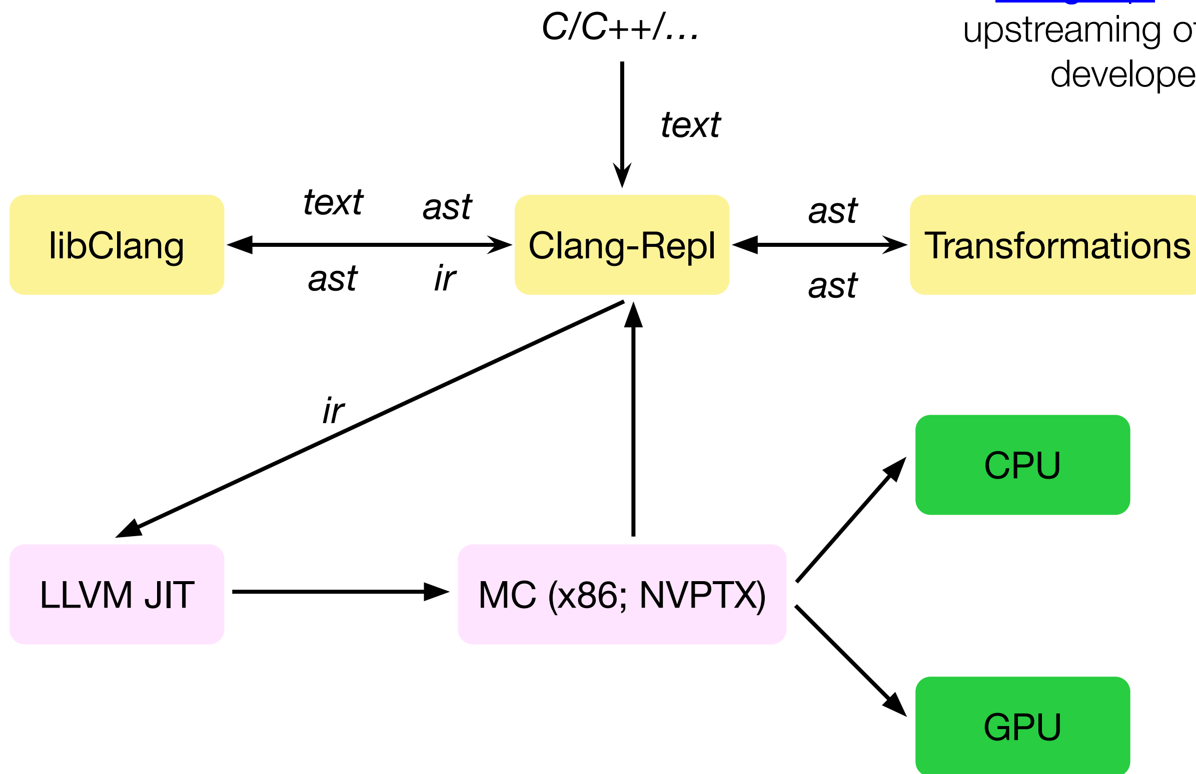
- We can split the translation unit into a sequence of **partial translation units** (PTU)
- Processing a PTU might extend an earlier PTU (template instantiation)



Clang-REPL: Data Flow

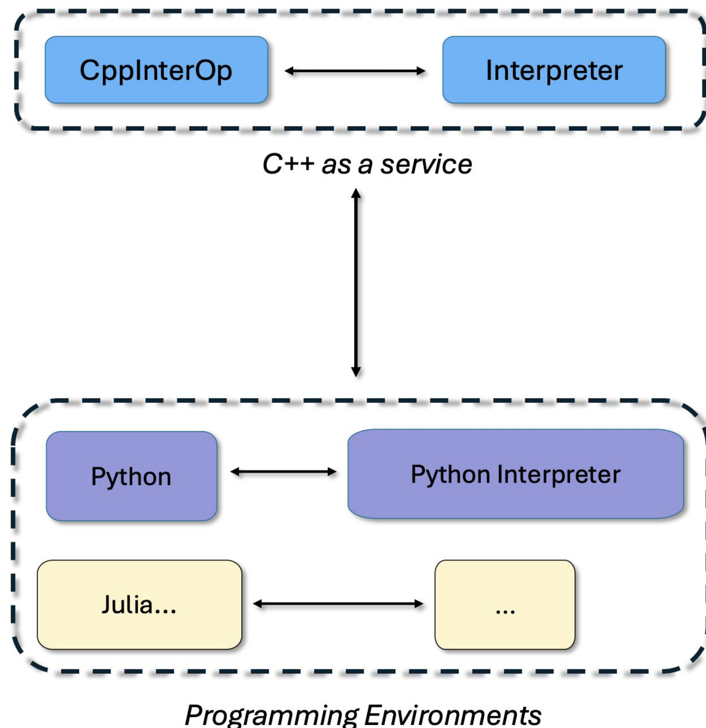
COMPILER C|R RESEARCH

[clang-repl](https://clang.llvm.org/docs/ClangRepl.html) is the result of continuous upstreaming of the [Cling](https://github.com/root-project/cling) C++ interpreter developed at CERN, to LLVM



CppInterOp

- Lightweight layer on top of LLVM/Clang
- Provides efficient, on-demand reflection and incremental JIT compilation capabilities
- Embeds Clang and LLVM as libraries in your framework, in a backward compatible way
- Supports downstream tools with reflection API for interactive C++ and language interoperability
- Works with C/C++ libraries from your favourite package manager



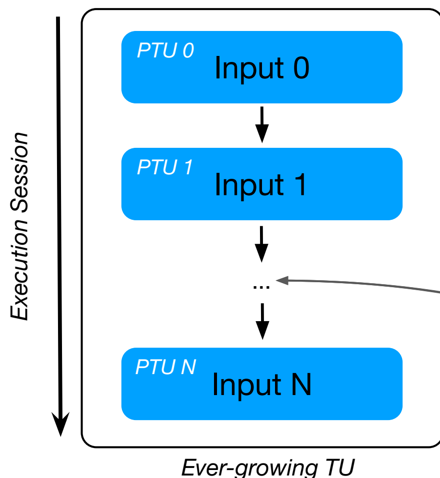
https://cppinterop.readthedocs.io/
https://github.com/compiler-research/CppInterOp

CppInterOp API

```
Cpp::Declare(R"(
        template<typename T>
        struct S {
          bool operator<(T &a) { return 0 < a; }
        };
    )", Decls);
ASTContext& C = Interp->getCI()->getASTContext();
auto Instance1 = Cpp::InstantiateTemplate(Decls[0],
                 {C.IntTy.getAsOpaquePtr()}, 1);
auto* obj = Cpp::Construct(Instance1);
std::vector<TCppFunction_t> ops;
Cpp::GetOperator(Instance1, Cpp::Operator::OP_Less, ops);
Cpp::JitCall FCI_Add = Cpp::MakeFunctionCallable(ops[0]);
int a = 5;
bool result = false;
void* args[1] = {(void*)&a};
FCI_Add.Invoke(&result, {args, /*arg_count=*/1}, obj);
assert(result == true);
```

Runtime Template Instantiation

```cpp
Cpp::Declare(R"(
        template<typename T>
        struct S {
          bool operator<(T &a) { return 0 < a; }
        };
    )", Decls);
ASTContext& C = Interp->getCI()->getASTContext();
auto Instance1 = Cpp::InstantiateTemplate(Decls[0],
                      {C.IntTy.getAsOpaquePtr()}, 1);
```



Specialized Decl

TemplateDecl

Cpp::InstantiateTemplate

TemplateArgInfo -> llvm::SmallVector<TemplateArgument>

Function

Variadic

Class

clang::Sema

DeduceTemplateArguments

InstantiateFunctionDefinition

CheckVarTemplateId

CheckTemplateId

RequireCompleteType

QualType

FunctionDecl

CXXRecordDecl

JitCall: Bridging compiled and interpreted code

COMPILER C|R RESEARCH

```
// Instance1 is an entity that lives in the interpreter
  auto* obj = Cpp::Construct(Instance1);
  std::vector<TCppFunction_t> ops;
  Cpp::GetOperator(Instance1, Cpp::Operator::OP_Less, ops);
  Cpp::JitCall FCI_Add = Cpp::MakeFunctionCallable(ops[0]);
  int a = 5;
  bool result = false;
  void* args[1] = {(void*)&a};
  FCI_Add.Invoke(&result, {args, /*arg_count=*/1}, obj);
  assert(result == true);
```

codegen

invoke

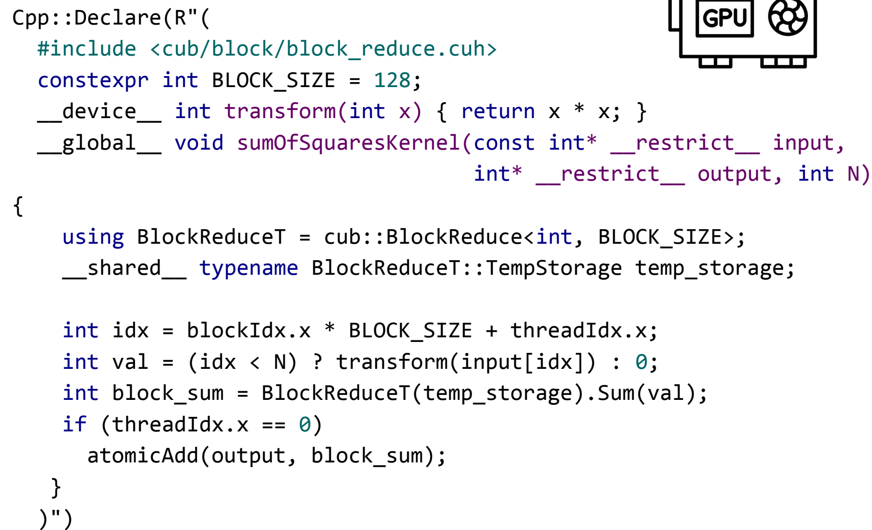
```
extern "C" void __jc_1(void* obj, unsigned long
nargs, void** args, void* ret)
{
   if (ret) {
      new (ret) (bool)
(((S<int>*)obj)->operator<((int&)*(int*)args[0]));
      return;
   }
   else {
//..
```

Host-Device JIT compilation

CppInterOp is based on Clang and the LLVM JIT

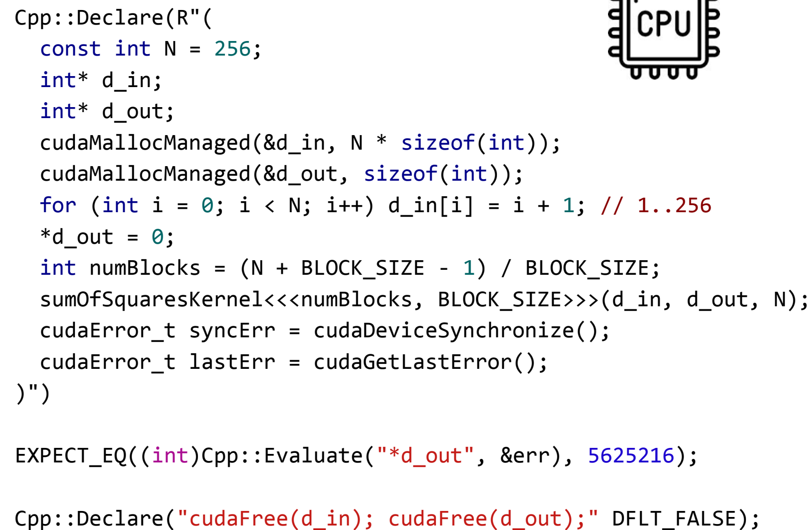
Supports hardware accelerators or parallel programming API:

- Cpp::CreateInterpreter(“--std=c++23”, “**--cuda**”)



```
Cpp::Declare(R"(
  #include <cub/block/block_reduce.cuh>
  constexpr int BLOCK_SIZE = 128;
  __device__ int transform(int x) { return x * x; }
  __global__ void sumOfSquaresKernel(const int* __restrict__ input,
                                     int* __restrict__ output, int N)
{
    using BlockReduceT = cub::BlockReduce<int, BLOCK_SIZE>;
    __shared__ typename BlockReduceT::TempStorage temp_storage;

    int idx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int val = (idx < N) ? transform(input[idx]) : 0;
    int block_sum = BlockReduceT(temp_storage).Sum(val);
    if (threadIdx.x == 0)
      atomicAdd(output, block_sum);
   }
  )")
```

```
Cpp::Declare(R"(
  const int N = 256;
  int* d_in;
  int* d_out;
  cudaMallocManaged(&d_in, N * sizeof(int));
  cudaMallocManaged(&d_out, sizeof(int));
  for (int i = 0; i < N; i++) d_in[i] = i + 1; // 1..256
  *d_out = 0;
  int numBlocks = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
  sumOfSquaresKernel<<<numBlocks, BLOCK_SIZE>>>(d_in, d_out, N);
  cudaError_t syncErr = cudaDeviceSynchronize();
  cudaError_t lastErr = cudaGetLastError();
)")

EXPECT_EQ((int)Cpp::Evaluate("*d_out", &err), 5625216);

Cpp::Declare("cudaFree(d_in); cudaFree(d_out);" DFLT_FALSE);
```

OpenMP

CppInterOp is based on Clang and the LLVM JIT

Supports hardware accelerators or parallel programming API:

- Cpp::CreateInterpreter(“--std=c++20”, “**-fopenmp**”)

```
Cpp::Declare(R"(
typedef struct link_t {
    struct link_t* next;
    int data;
} link_t;

void list_push2(link_t** l, int data)
{
    link_t* link = (link_t*) malloc(sizeof(link_t));
    link->data = data;
    #pragma omp atomic capture
    {
        link->next = *l;
        *l = link;
    }
})")
```

```
Cpp::Declare(R"(
void list_traversal_test() {
    link_t* head = NULL;
    for (int i = 0; i < 10; ++i) list_push2(&head, i);
#pragma omp parallel
    {
#pragma omp single nowait
        {
            for (link_t* link = head; link; link = link->next)
{
#pragma omp task firstprivate(link)
                printf("Process %d on thread %d\n", link->data,
                       omp_get_thread_num());
            }
        }
    }
})")
```

Library Loading

COMPILER C|R RESEARCH

```
Cpp::LoadLibrary("mylib.so");
Cpp::Declare("#include \"mylib.h\"");
auto f = reinterpret_cast<int (*)(int, int)>(Cpp::GetFunctionPointer("add"));
assert(f(2, 3) == 5);
```

Symbol-guided

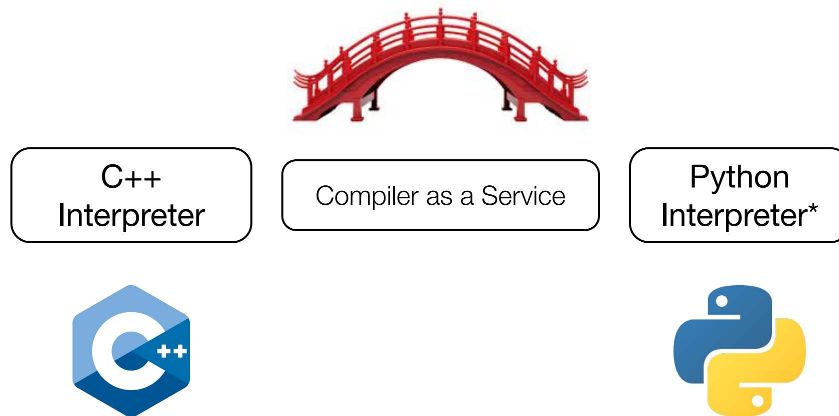
```
// register a search path for shared libraries
Cpp::AddSearchPath("/opt/myproject/lib");
std::string lib = Cpp::SearchLibrariesForSymbol("sum", /*system_search=*/false);

// lib == "/opt/myproject/lib/libMyMath.so"
Cpp::LoadLibrary(lib.c_str());
Cpp::Process(""); // force creation of ExecutionEngine
auto f = reinterpret_cast<int (*)(int, int)>(Cpp::GetFunctionAddress("sum"));
assert(f(2, 3) == 5);
Cpp::UnloadLibrary("MyMath"); // release library handle
```

Downstream Clients

COMPILER
C | R
RESEARCH

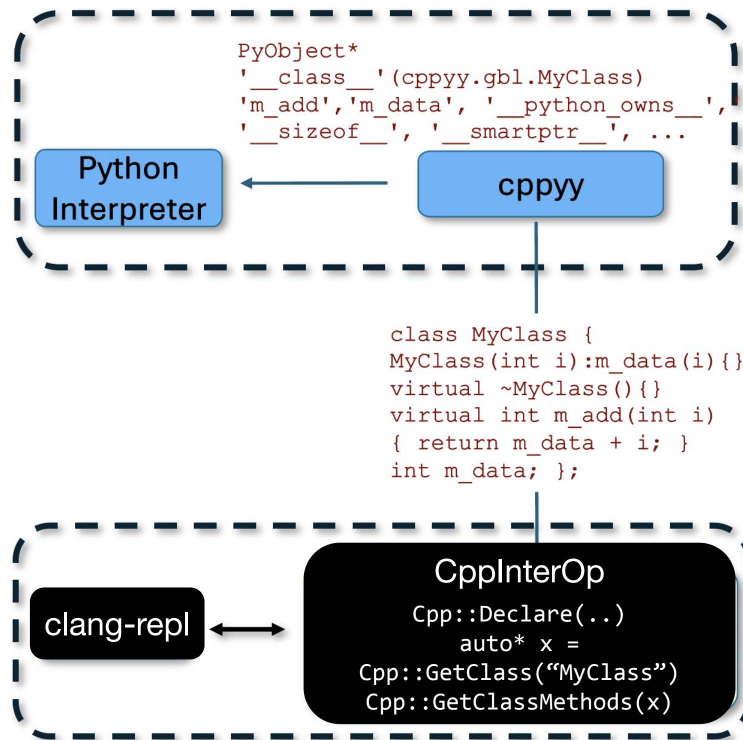
Bridging interpreted C++ with other environments



*And more

Automatic Python/C++ Interoperability

```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)
>>> cppyy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppyy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
>>> class PyMyClass(MyClass):
...     def add_int(self, i):  # python side override (CPython only)
...         return self.m_data + 2*i
...
>>> cppyy.cppdef("int callback(MyClass* m, int i) { return m->add_int(i); }")
True
>>> cppyy.gbl.callback(m, 2)             # calls C++ add_int
15
>>> cppyy.gbl.callback(PyMyClass(1), 2)  # calls Python-side override
5
>>>
```



The cppyy project is based on Cling: https://cppyy.readthedocs.io/en/latest/
Compiler-Research moved cppyy to **CppInterOp** and **clang-repl**
Releases on PyPI coming soon: https://github.com/compiler-research/CppJIT

Other programming languages

https://jank-lang.org/

https://github.com/jank-lang/jank

```
(ns jank.json
  (:include "fstream"
            "./json.hpp")
  (:refer-global :only [std.ifstream nlohmann.json.parse]
                 :rename {std.ifstream open-file
                          nlohmann.json.parse parse-json}))

(defn -main [& args]
  (let [file (open-file (cpp/cast std.string (first args)))
        json (parse-json file)]
    (println (.dump json 2))))
```

Dialect of Clojure built on LLVM, extends CppInterOp for C++ Interoperability (AOTCall, extra predicates and transformations)

First-class support for C++ includes, bringing C++ globals into a jank namespace and renaming them.

```
julia> import CppInterOp as Cpp

julia> using Test

julia> I = Cpp.create_interpreter()
CppInterOp.Interpreter(Ptr{CppInterOp.LibCppInterOp.CXInterpreterImpl}(0x000000001c2c0c60))

julia> Cpp.process(I, "int x = 1 + 1;") # PTU_1
true

julia> Cpp.evaluate(I, "x")           # creates PTU_2
2

julia> Cpp.undo(I, 2)
true

julia> Cpp.process(I, "float x = 42.0;")
true

julia> Cpp.evaluate(I, "x")
42.0f0
```

Thanks to Yupei Qi (@Gnimuc)!

> pkg add https://github.com/Gnimuc/CppInterOp.jl

xeus-cpp

COMPILER C|R RESEARCH

Jupyter kernel for C++ based on the native implementation of the Jupyter protocol xeus

Successor to the xeus-cling project

Leverages CppInterOp for clang-repl and CaaS facilities

xeus-cpp-lite extends this model to the browser with WebAssembly and JupyterLite

Thanks to the QuantStack team!

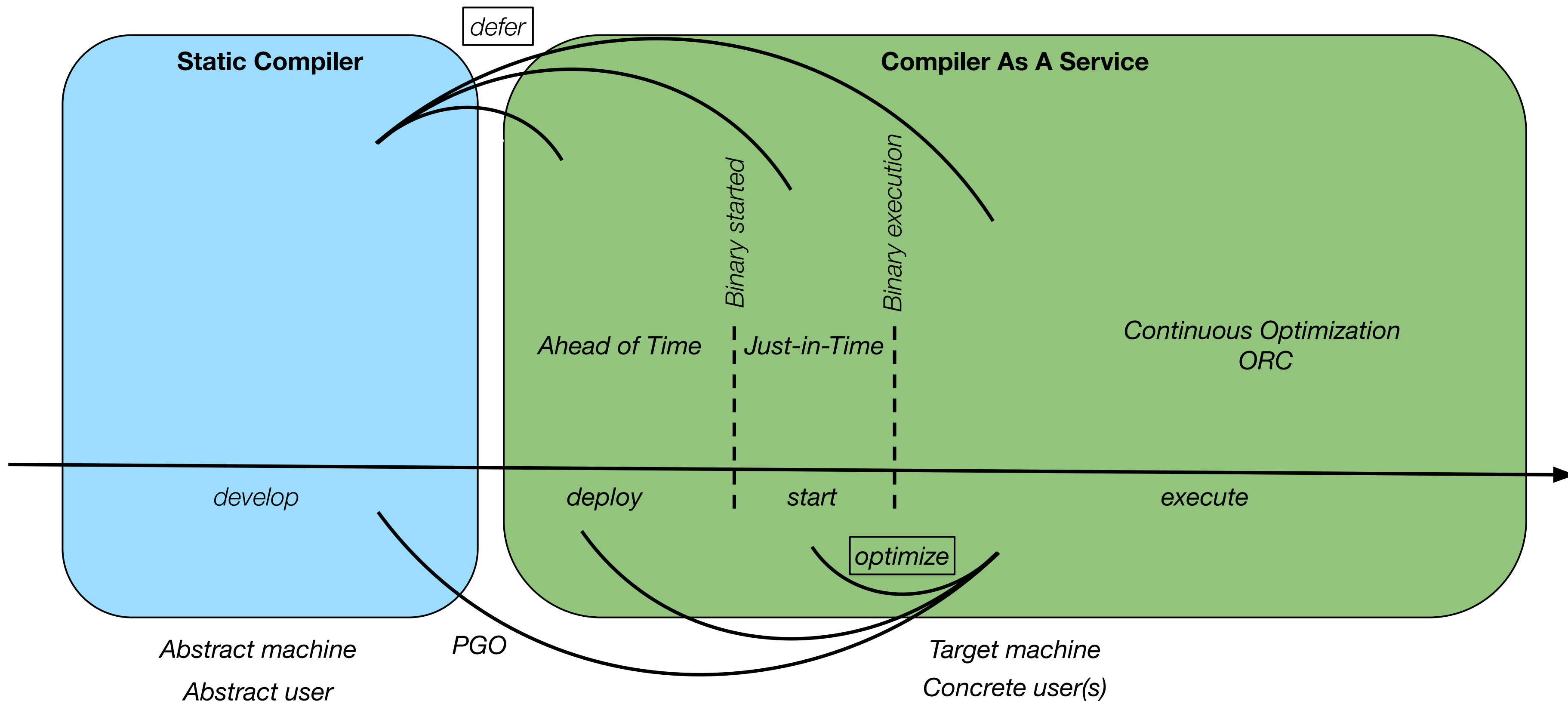
https://github.com/compiler-research/xeus-cpp

Demo

COMPILER C|R RESEARCH

- Interactive C++ notebooks with xeus-cpp
  - Sub-interpreters with different C++ standards, OpenMP
  - Running CUDA in notebooks
- C++ execution in the browser with xeus-cpp-lite and WebAssembly
  - https://compiler-research.org/CppInterOp/lab/index.html
- Putting it all together – Exploratory Programming Workflow

Summary: CaaS Interaction And Opportunities



COMPILER C|R RESEARCH

Thank You!