PRINCETON
UNIVERSITY

cppyy

CppInterOp: Improving language bindings by bringing Cppyy closer to LLVM

Aaron Jomy (Princeton University)
Supervisor: Dr. Vassil Vassilev (CERN/Princeton)

BRIEF INTRO

- Cppyy
  An automatic C++ - Python runtime bindings generator which provides the user with the C++ feature set
- Cling
  An interactive C++ interpreter, built on LLVM and Clang
  Used in Cppyy's(upstream) backend
- Clang-REPL
  A generalization of Cling in LLVM - supports interactive programming for C++ in a read-evaluate-print-loop (REPL) style

cppyy

MOTIVATING EXAMPLE

```
Eigen::Vector2d a;
Eigen::Vector3d b;

template <typename VectorType>
class EigenOperations {

static VectorType PerformOperations(const VectorType& v) {
VectorType result = v;

result.normalize(); // eigen vector normalisation

double dot = v.dot(v); // dot product
for (int i = 0; i < result.size(); i++) {
        result[i] += dot;
}
double squaredNorm = v.squaredNorm(); // vector squared norm
for (int i = 0; i < result.size(); i++) {
        result[i] -= squaredNorm;
}

return result;
}
};
```

We utilize a C++ utility header that uses the Eigen templated library for vector operations.

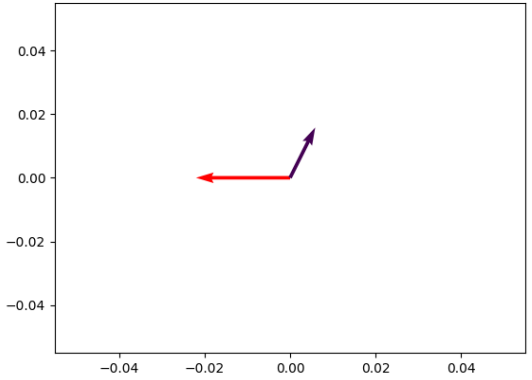
MOTIVATING EXAMPLE

```
import cppyy

cppyy.include("eigen_utils.h")
Cpp = cppyy.gbl

a = Cpp.a
b = Cpp.b

for i in range(len(a))
a[i] = i - 1

res2d = Cpp.EigenOperations["Eigen::Vector2d"].PerformOperations(a)
res3d = Cpp.EigenOperations["Eigen::Vector3d"].PerformOperations(b)

np_res2 = np.array(list(res2d))
np_res3 = np.array(list(res3d))

plt.quiver(*origin, *np_res2, color=['r'], scale=5)
plt.quiver(*origin, *np_res3, color=['b'], scale=5)

plt.savefig("test.jpg")
plt.show()
```

Results on Python Interpreter

```
Vector :  A , Dimensions :  2
-1.0
0.0
Vector :  B , Dimensions :  3
0.26726124191242384
0.5345224838248495
0.8017837257372733
```

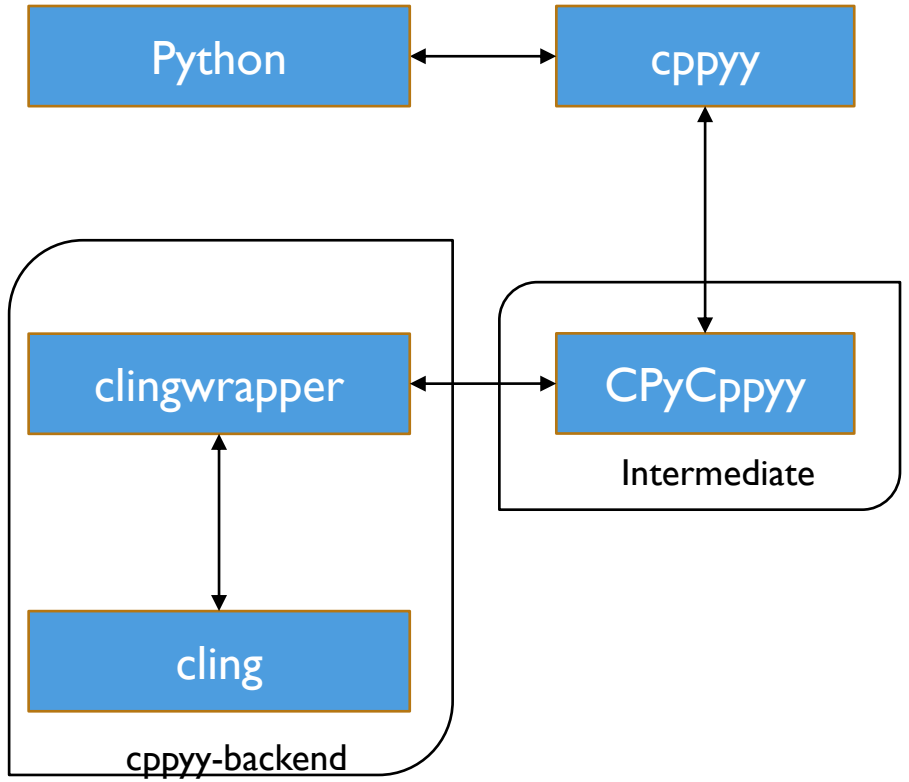
Matplotlib:



DIVING INTO CPPYY INTERNALS

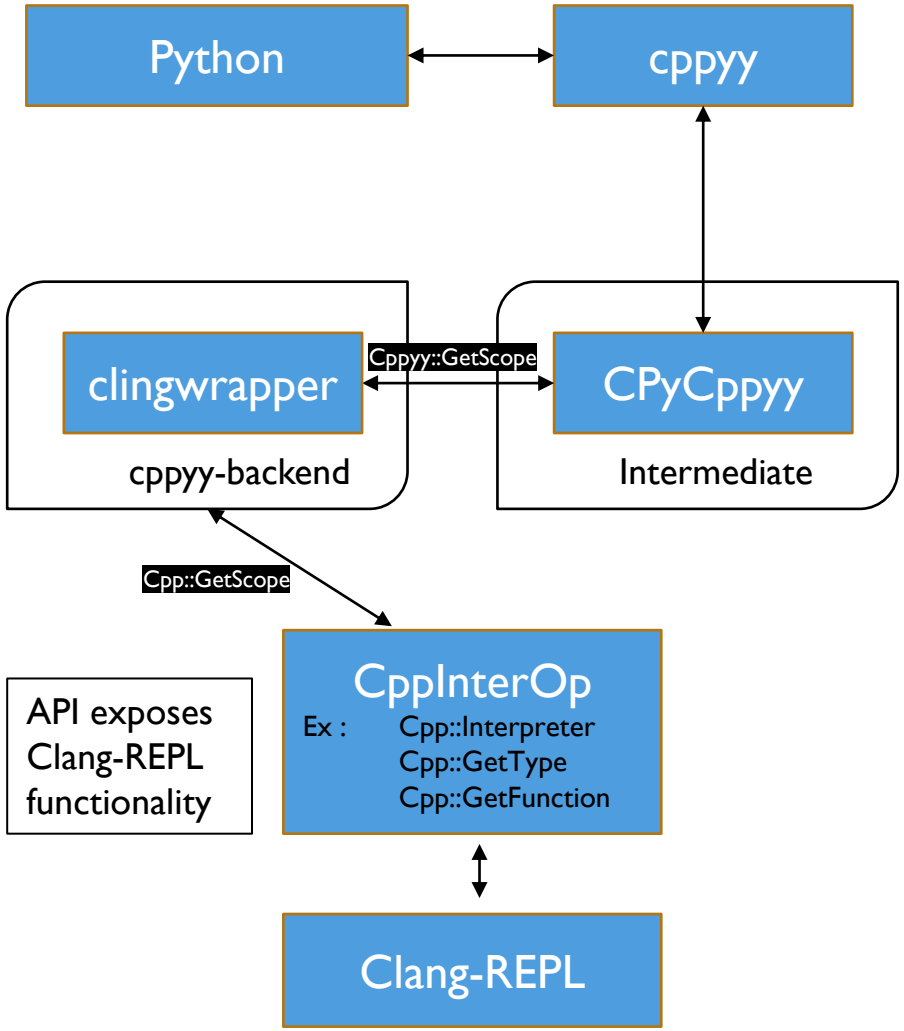
- Four PyPA packages involved in a full installation
- The cppyy module is a frontend that requires an intermediate (Python interpreter dependent) layer, and a backend
- CPyCppyy : uses Python C-API to bridge the information from Python and Cling. Constructs Python proxies (CppOverload, CppMethod, CppClass) of C++ entities.
- The cppyy-backend module contains two areas:
  - A patched copy of cling
  - Wrapper code (clingwrapper)

Cppyy upstream based on Cling



DIVING INTO CPPYY INTERNALS

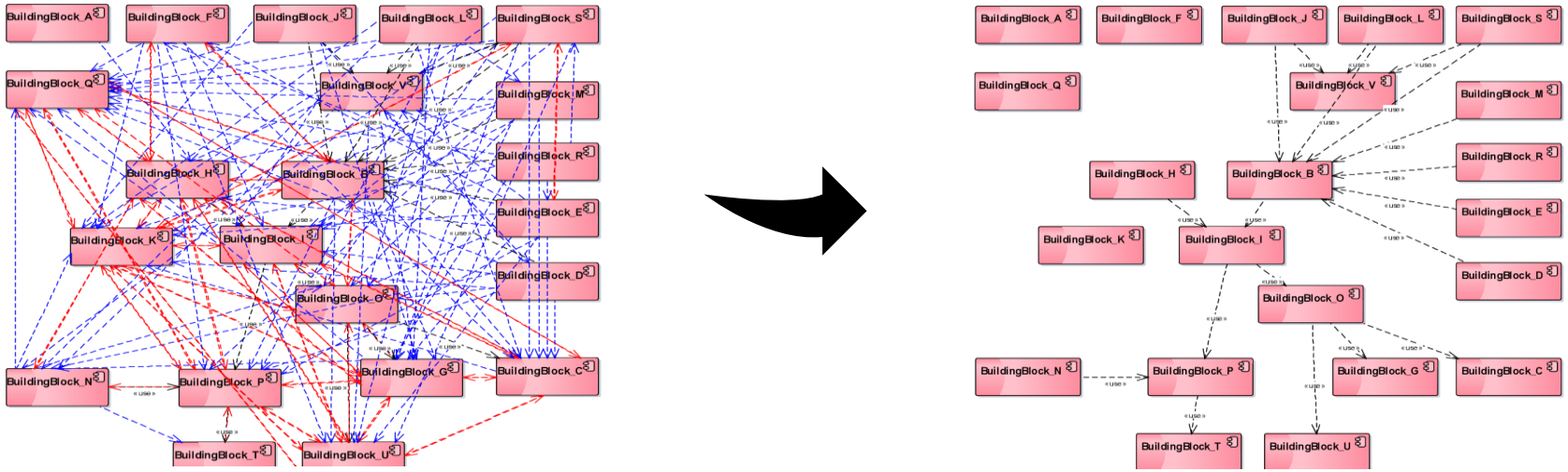
- CppInterOp allows Cppyy to use LLVM's Clang-REPL as a runtime compiler
- This avoids the string parsing logic used with the current Cling based cppyy-backend
- Opens up more C++ features that can be used by Cppyy users
- Lower dependencies leads to performance improvement
- CppInterOp unit tests verify the API that is used for proxy creation, lookups, function reflection, etc



ADVANTAGES OF TRANSITIONING TO CLANG-REPL

Simplified codebase:

- Compiler features are abstracted to the InterOp layer interfaces
- Easier to extend: Modular development of tests/features
- Reduced string manipulation with parsed code



ADVANTAGES OF TRANSITIONING TO CLANG-REPL

Simplified codebase

Example: Cppyy::GetMethodTemplate

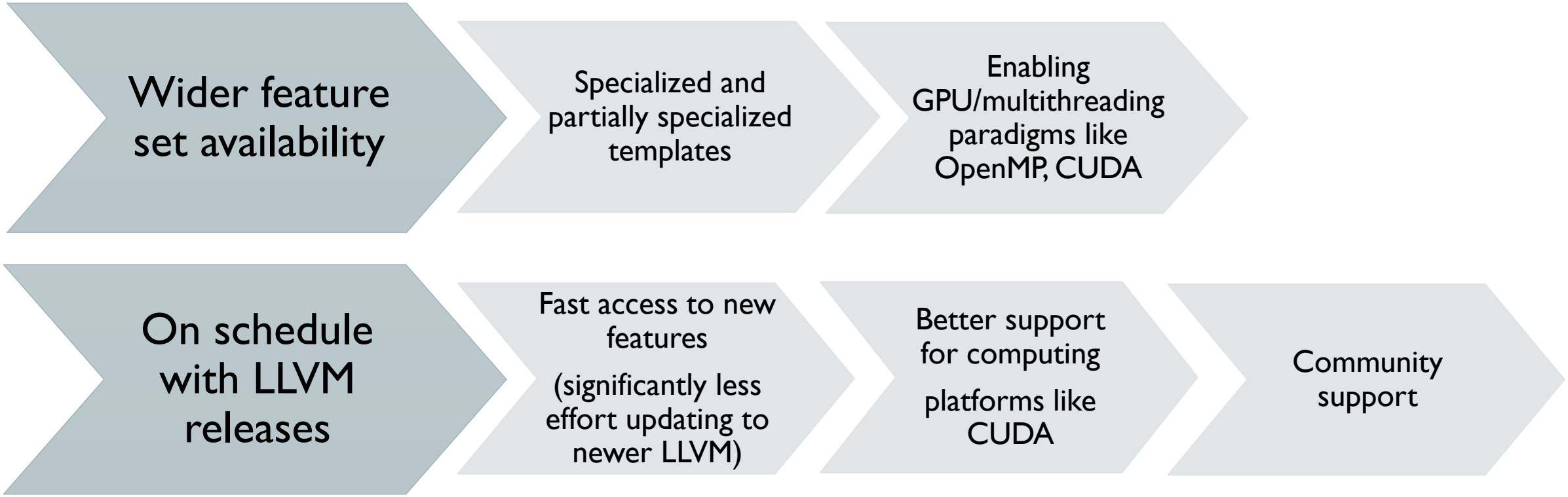
Cppyy Upstream - clingwrapper

```
TFunction * func = nullptr;
ClassInfo_t * cl = nullptr;
if (scope == (cppyy_scope_t) GLOBAL_HANDLE) {
  func = gROOT -> GetGlobalFunctionWithPrototype(name.c_str(), proto.c_str());
  if (func && name.back() == '>') {
if (!template_compare(name, func -> GetName()))
      func = nullptr; // happens if implicit conversion matches the overload
  }
} else {
  TClassRef & cr = type_from_handle(scope);
  if (cr.GetClass()) {
    func = cr -> GetMethodWithPrototype(name.c_str(), proto.c_str());
    if (!func) {
      cl = cr -> GetClassInfo();
TCppIndex_t nbases = GetNumBases(scope);
      for (TCppIndex_t i = 0; i < nbases; ++i) {
        TClassRef & base = type_from_handle(GetScope(GetBaseName(scope, i)));
        if (base.GetClass()) {
          func = base -> GetMethodWithPrototype(name.c_str(), proto.c_str());
          if (func) break;
        }
      }
    }
  }
```

With CppInterOp

```
Cppyy::TCppMethod_t Cppyy::GetMethodTemplate(
TCppScope_t scope, const std::string& name, const std::string& proto)
{
        Cpp::GetMethodTemplate(scope, name);
}

TCppFunction_t GetTemplatedMethod(const std::string& name,
TCppScope_t parent, const std::string& filter)
{
DeclContext *Within = 0;
DeclContext::decl_iterator decl;
if (parent) {
auto *D = (Decl *)parent;
Within = llvm::dyn_cast<DeclContext>(D);
}

auto *ND = Cpp_utils::Lookup::Named(&getSema(), name, Within); if
((intptr_t) ND == (intptr_t) 0)
return nullptr;

if ((intptr_t) ND != (intptr_t) -1)
return (TCppFunction_t)(ND->getCanonicalDecl());
}
```

ADVANTAGES OF TRANSITIONING TO CLANG-REPL



ADVANTAGES OF TRANSITIONING TO CLANG-REPL

InterOp Unit Tests

compiler features tested on InterOp layer

```
TEST(CUDATest, CUDAH) {
if (!HasCudaSDK())
return;

Cpp::CreateInterpreter({}, {"--cuda"});
bool success = !Cpp::Declare("#include <cuda.h>");
EXPECT_TRUE(success);
}

TEST(CUDATest, CUDARuntime) {
if (!HasCudaSDK())
return;

EXPECT_TRUE(HasCudaRuntime());
}

EXPECT_EQ(Cpp::GetTypeAsString(Cpp::GetType("struct")),"NULL TYPE");
EXPECT_EQ(Cpp::GetTypeAsString(Cpp::GetType("char")), " char");
```

```
TEST(TypeReflectionTest, GetSizeOfType) {
std::vector<Decl *> Decls;
std::string code = R"(
struct S {
int a;
double b;
};

char ch;
int n;
S s;
)";

GetAllTopLevelDecls(code, Decls);

EXPECT_EQ(Cpp::GetSizeOfType(Cpp::GetVariableType(Decls[1])), 1);
EXPECT_EQ(Cpp::GetSizeOfType(Cpp::GetVariableType(Decls[2])), 4);
EXPECT_EQ(Cpp::GetSizeOfType(Cpp::GetVariableType(Decls[3])), 16);
```

CURRENT PLAN

Certain failure reasons and bugs have been identified:

- Template instantiation
- Reflection layer failures
- Invalid scope requests from cppyy make its way through the reflex tooling in CppConstructor in CPyCppyy.

  These don't work with Cpp::GetScope in CppInterop where lookups don't accept ns::class

  Also responsible for failure of return types in Numba

```
return_type=cpp2numba(ol._func.__cpp_reflex__(cpp_refl.RETURN_TYPE))
```

Goal - Stabilize the clang-repl build

Primary focus is tests with high fail/pass ratio and tests with common failure sources (interfaces, memory leaks, etc)

Eventually make CppInterOp a part of LLVM upstream

Thank You!

Aaron Jomy