Calling C++ libraries from a D-written DSL: A cling/cppyy-based approach

Compiler as a Service project @ Princeton University

Alexandru Militaru

alexandru.cmilitaru@gmail.com

4 February, 2021

Symmetry INVESTMENTS

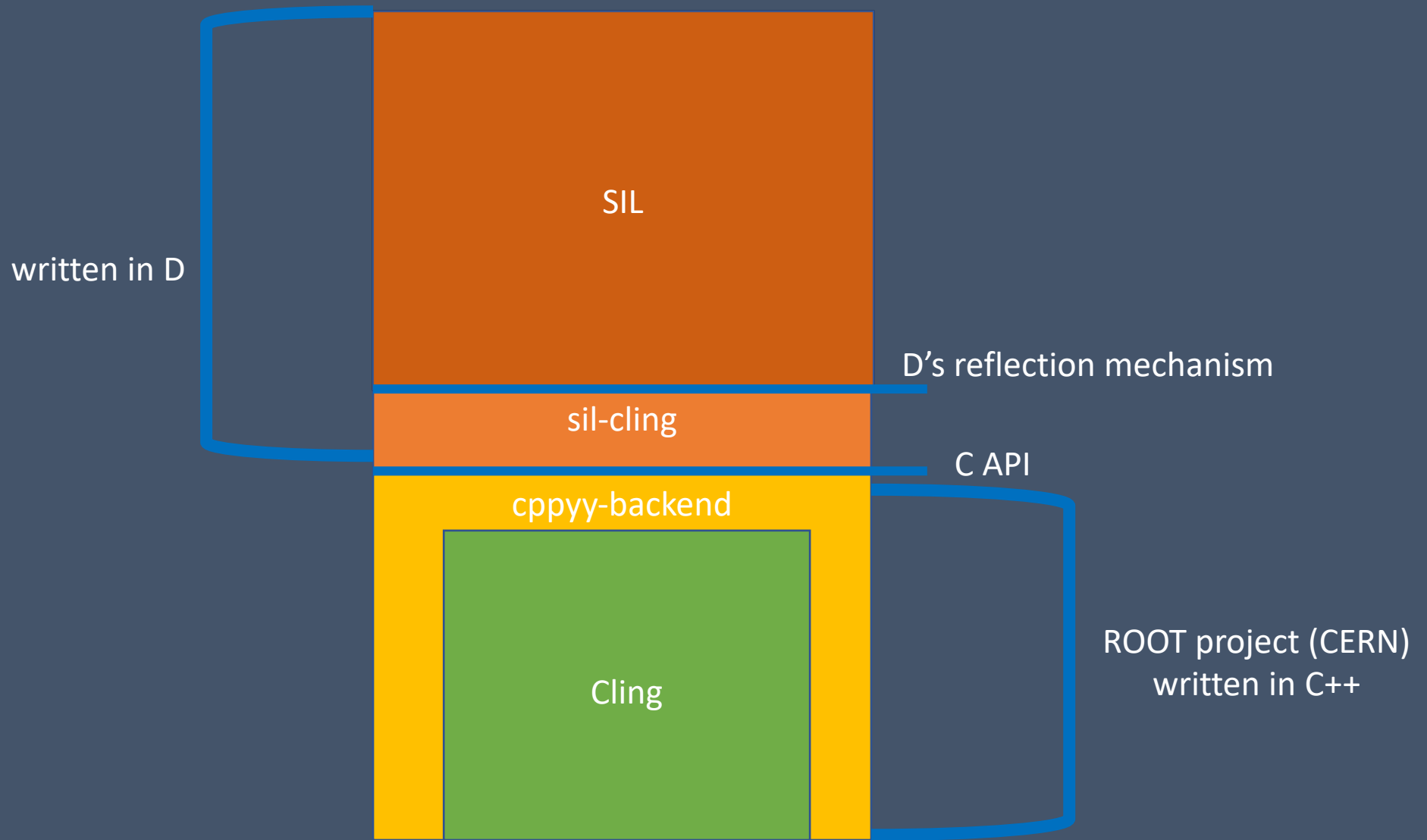
Symmetry Integration Language (SIL)

- D-based domain specific language of functional flavour
- designed from the ground up to be easily interoperable with other languages and systems

sil-cling

- SIL plugin that allows transparent calling of C++ libraries

sil-cling: Architecture



sil-cling: Interface with cppyy

- binds with cppyy through the direct inclusion of the latter's C header using dpp

```d
//cling.dpp

#include "capi.h"  // cppyy's C header

// D code ↓
import std.string : fromStringz, toStringz;

string resolveName(string cppItemName)
{
    import core.stdc.stdlib : free;
    // Calling cppyy_resolve_name ↓
    char* chars = cppyy_resolve_name(cppItemName.toStringz);
    string result = chars.fromStringz.idup;
    free (chars);
    return result;
}
```

```
→  ~ dub run dpp -- cling.dpp --keep-d-files -c
```

Using Cling to interact with...Cling (I)

```
 1 // cling.dpp
 2
 3 static this ()
 4 {
 5     compile (`#include <stdio.h>
 6                 struct ClingDStatics {
 7                   void addIncludePath (const char *path) {
 8                       gInterpreter->AddIncludePath (path);
 9                   }
10                 };`);
11 }
```

Using Cling to interact with...Cling (II)

```
 1 void addIncludePath (const string path) {
 2
 3     // Scope wraps cppyy_scope_t
 4     auto sc = Scope ("ClingDStatics");
 5
 6     // Class wraps cppyy_object_t - creates ClingDStatics object
 7     // Type wraps cppyy_type_t
 8     auto obj = Class.construct (Type(sc.handle));
 9
10     // Get cppyy_method_t
11     auto meth = sc.method ("void", "addIncludePath", "const char* path");
12
13     // ClingDStatics.addIncludePath(path)
14     // function template - calls cppyy_call_* depending on the template param
15     obj.call!void (meth, fullPath(path).toStringz);
16 }
```

sil-cling: Interface with SIL

- exposes wrappers of C++ entities to SIL through D's reflection mechanism
- two core types:
  - CPPNamespace
  - ClingObj
- both CPPNamespace and ClingObj classes have a data member that holds a reference to their associated Scope object

sil-cling: Calling Interface

- object construction, method calling, and function calling – all are done through the same interface

```
Variable call(T, string, A...)(T obj, string funcName, A args_seq)
    if (is (T == ClingObj) || is(T == CPPNamespace))
{
    .....
}
```

- the procedure is divided into three separate phases:
  1. Overload resolution
  2. Argument conversion
  3. Calling

sil-cling: Overload Resolution

- for a given object or namespace, fetch all the method/function overloads with the given name

- it's a match:
  - a method overload with the right number of arguments
  - each SIL Variable provided must have a valid type conversion to its corresponding parameter

- two iterations:
  - first, we allow only 'exact' conversions (e.g. long Variable → C++ long)
  - then we include 'lossy' conversions too (e.g. long Variable → C++ int)

```
struct TypeCoercion {
    string    cppType;
    bool      delegate(Variable) canCoerce;
    Parameter delegate(Variable) coerce;
}
// Rule for coercion to C++ char*
TypeCoercion (
    "char*", //cppType
     v => v.kind == KindEnum.string_,  //rule
     v => Parameter(v.get!string.toStringz) //ClingArg
);
```

rguments hold a

the C++ object

er

- SIL primitive type -> must match one of the predefined TypeCoercion rules

sil-cling: Calling

- examine the return type of the selected overload and figure out to what SIL type should it be converted

- distinguish between C++ primitive types, types that should be wrapped as ClingObjs, and types not supported yet

```
1  Variable call_impl (Method meth, ClingArgs ca)
2  {
3      ...
4      switch (meth.resultType)
5      {
6          case "void":
7              return this.call_v (meth, ca);
8          case "int":
9              return this.call_i (meth, ca);
10         case "int&":
11             return this.call_ir (meth, ca);
12         case "long":
13             return this.call_l (meth, ca);}}
14         ...
15         default:
16             // if ClingObj
17                 // do what it takes
18             // else assert(0, "Type not found);
19     }
20     ...
21 }
```

sil-cling: Data Members

- involves an offset calculation

```
// Handles primitive types only
void set_impl(E, T) (E entity, DataMember dm, T data)
{
    // If the data member is static
    // or is the data member of a namespace,
    // then the its offset should be taken as its addres
    void* ptr = cast(void*) dm.offset;

    static if (is(E == ClingObj))
    {
        if (!dm.isStatic())
            ptr = entity._class.handle + dm.offset;
    }

    T* ptrToDm = cast(T*) ptr;
    *ptrToDm = data;
}
```

sil-cling: What's next?

- automatic instantiation of templates
- direct conversions to/from SIL arrays and std::vector

```
 1 import * from silcling
 2
 3 cppCompile(
 4     "namespace N {
 5         template <class T>
 6         T justRet (T a) {
 7             return a;
 8         }
 9     }")
10 nspace = cppNamespace("N")
11 // instantiate template by hand
12 nspace.tempInst("justRet<int>", "int")
13 enforce(nspace.justRet(8) == 8, "")
```

sil-cling: Summary

- a SIL plugin that allows transparent calling of C++ libraries
- built using cling and cppyy
- works with Boost.Asio, dlib, Xapian, etc.