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 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**: D’s reflection mechanism
- **sil-cling**: C API
- **cppyy-backend**: Written in D
- **Cling**: ROOT project (CERN) written in C++
sil-cling: Interface with cppyy

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

```dub
//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)

```c
// cling.dpp

static this ()
{
    compile ("#include <stdio.h>
        struct ClingDStatics {
            void addIncludePath (const char *path) {
                gInterpreter->AddIncludePath (path);
            }
        };
    
```
Using Cling to interact with…Cling (II)

```cpp
void addIncludePath (const string path) {

    // Scope wraps cppyy_scope_t
    auto sc = Scope ("ClingDStatics");

    // Class wraps cppyy_object_t - creates ClingDStatics object
    // Type wraps cppyy_type_t
    auto obj = Class.construct (Type(sc.handle));

    // Get cppyy_method_t
    auto meth = sc.method ("void", "addIncludePath", "const char* path");

    // ClingDStatics.addIncludePath(path)
    // function template - calls cppyy_call_* depending on the template param
    obj.call!void (meth, fullPath(path).toStringz);
}
```
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

```cpp
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 $\rightarrow$ C++ long)
  - then we include ‘lossy’ conversions too
    - (e.g. long Variable $\rightarrow$ C++ int)
Ød depends on whether the SIL Variables provided as arguments hold a SIL primitive type or a ClingObj:

- ClingObj -> string comparisons between the type of the C++ object that it wraps and the type of the overload's parameter

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

```cpp
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
);
```
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

```cpp
Variable call_impl (Method meth, ClingArgs ca) {
  ...
  switch (meth.resultType) {
    case "void":
      return this.call_v (meth, ca);
    case "int":
      return this.call_i (meth, ca);
    case "int&":
      return this.call_ir (meth, ca);
    case "long":
      return this.call_l (meth, ca);}}
  ...
  default:
    // if ClingObj
    // do what it takes
    // else assert(0, "Type not found);
}
```
sil-cling: Data Members

- involves an offset calculation

```c
// 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 address.
    void* ptr = cast(void*) dm.offset;

    static if (is(E == ClingObj))
    {
        // If the data member is static.
        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

```cpp
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)
11 nspace = cppNamespace("N")
12 // instantiate template by hand
13 nspace.tempInst("justRet<int>", "int")
14 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.