P3010R0 - Using Reflection to Replace a Metalanguage for Generating JS Bindings

Reply-to: Dan Katz
Last Updated: October 12, 2023
Audience: SG7

Abstract

"ROB" is a proprietary Bloomberg C++ preprocessor language, whose raison d'être is to surface natively implemented classes through JavaScript APIs with almost no boilerplate code. To do its job, the ROB compiler generates wrappers to translate between native C++ and JavaScript types and conventions, as well as "type descriptor" structs that cannot yet be fully generated using ISO C++ (modulo macros or significant boilerplate). ROB components play central roles in server-side JavaScript applications accessed daily by hundreds of thousands of external clients, as well as internal users.

This paper describes an experimental framework, affectionately named `rob2`, that permits the replacement of ROB components with pure C++ classes by leveraging value-based reflection as provided by the lock3 implementation of P2320 ("*The Syntax of Static Reflection*"). Reflection capabilities that were found to be important are highlighted, as well as implementation challenges and frustrations encountered along the way.

Since the motivation for `rob2` was to exercise the value-based reflection model, and to provide a thought provoking proof of concept for internal users of ROB, no attempt was made to engineer a "complete" solution: attentive readers will not find it difficult to identify cases that would fail to compile, or do "the wrong thing" given the described implementation (e.g., overloading of class member functions).

Background

ROB is most heavily used by an application server originally written in C, which maintains a runtime type registry of all native types surfaced through JavaScript. The chief task of ROB is to instantiate and register a `struct RobTypeDescriptor` which captures enough metadata to permit runtime reflection of type names, constructors, member and class member functions, properties (i.e., getters, setters), inheritance relationships, etc., as well as to generate the code for marshaling between native and JavaScript types. A separate component of the application server (beyond the scope of this paper) is responsible for querying the type registry and constructing the corresponding JavaScript API objects.

A brief introduction to ROB

Code generation with ROB

Suppose there is some proprietary server-side API

```
string_view current_user();
```

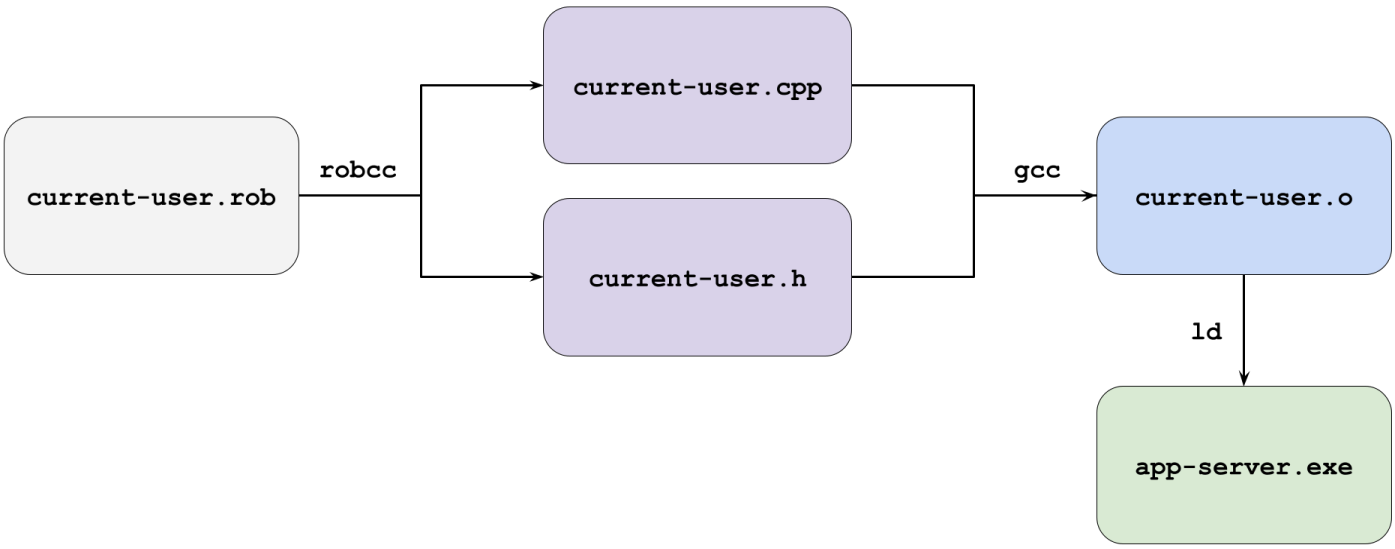
which returns the username for the request that is being served. An audience of application developers has requested that this function be callable from server-side JavaScript. With the help of ROB, this is a breeze.

We write a ROB component, and name it `current-user.rob`:

```
class CurrentUser extends RobBaseObject {
    public static const char *username() {
        string_view username = current_user();
        return strdup(username.data());
    }
}
```

which looks a lot like a C++ class! Don't be fooled - this won't be compiled directly. Instead, it will be preprocessed by the `robcc` transpiler, which will:

1. Parse the `current-user.rob` file.
2. Generate corresponding header and definition files (i.e., `current-user.h`, `current-user.cpp`).
3. Link the resulting `current-user.cpp` file into the application server.



Surfacing ROB types through JavaScript

From this simple component, 500+ lines of boilerplate code may be generated. At its heart though, the following declaration is found in `current-user.h`:

```
const char *current_user_username();
```

and the following definition is found in `current-user.cpp`:

```
const char *current_user_username() {
    string_view  username = current_user();
    const char  *result   = strdup(username.data());
    return result;
}
```

The C heritage of the application server shines proudly: No C++ class is generated at all! A set of C-accessible APIs are produced instead. Nevertheless, the type system plugged into by ROB-generated code supports many object-oriented features (e.g., inheritance, information hiding, virtual dispatch). **`robcc`** makes much of this possible by defining a static global **`RobTypeDescriptor`** struct within **`current-user.cpp`**, and populating it with information describing the **`CurrentUser`** ROB component.

```
static RobTypeDescriptor rob_type_current_user = {
    ROB_TYPE_ROB_BASE_OBJECT, /* parent type */
    …
    "CurrentUser",            /* class name */
    …
    rob_type_vtable           /* virtual table */
};
```

A generated static initializer registers this struct with the application server's type registry before the program enters **`main`**, thus permitting the runtime discovery of **`CurrentUser`** and its methods and properties. Other components of the application server will leverage this to surface an object named **`CurrentUser`** through the JavaScript runtime; it will have a property **`CurrentUser.username`**, which may be called as a function taking no arguments and returning a string.

```
> CurrentUser.username()
"katzdm"
```

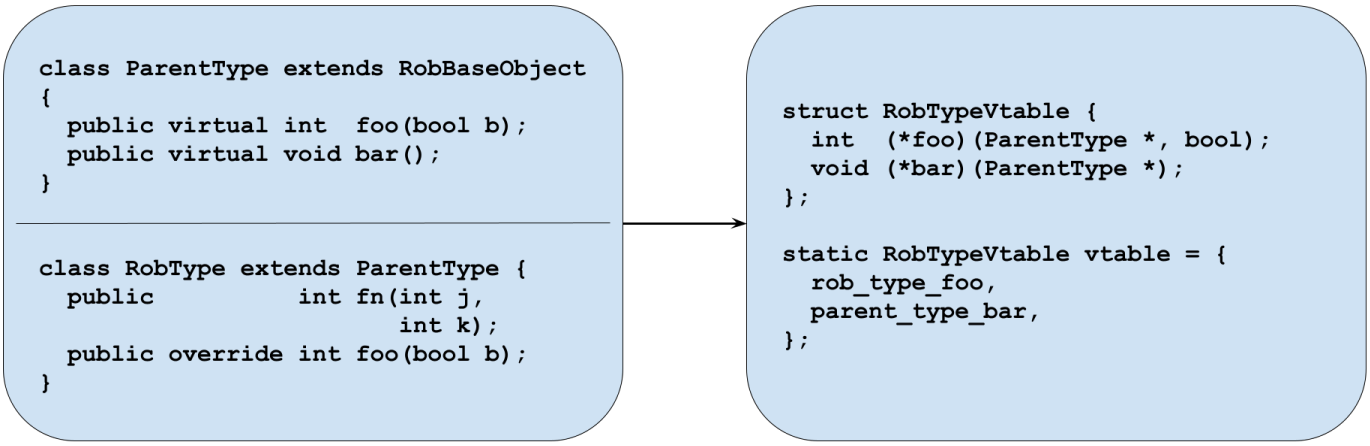
ROB virtual tables

The initializer for the above **`RobTypeDescriptor`** includes a value **`rob_type_vtable`**, which is worth briefly exploring in more depth.

For every ROB component **`RobType`**, the **`robcc`** transpiler generates:

1. A C compatible **`struct RobType`** having any data members defined by the component.
2. A corresponding C compatible **`struct RobTypeVtable`**.

`RobTypeVtable` is the *virtual table struct* corresponding to **`RobType`**, and it contains a function pointer data member corresponding to every declared *or* inherited virtual member function belonging to **`RobType`**.



Thus, for every virtual member function `fn` belonging to `RobType` (whether declared or inherited), there is a function pointer data member *also named* `fn` belonging to `RobTypeVtable`, such that if `RobType::fn` has the signature

```
Result (CLS::*)(Arg1, …, Argn);
```

then `RobTypeVtable::fn` has the type

```
Result (*)(CLS *, Arg1, …, Argn);
```

Furthermore, `RobTypeVtable::fn` will be initialized to the function corresponding to the *most derived* implementation of the corresponding virtual function.

Generating and initializing this data structure proved to be one of the more interesting problems encountered while implementing `rob2`.

Shortcomings of ROB

Although ROB succeeds in saving engineers from manually wiring native code through the JavaScript runtime, it nevertheless leaves a lot to be desired.

- Writing ROB requires a high learning curve and cognitive overhead on top of the already complex C++ language: A rich and exciting world of new and novel footguns awaits.

- The amount of boilerplate generated by `robcc` is very large: the `.h` and `.cpp` files generated from `.rob`'s are arduous to dig through and all but illegible to non-experts (e.g., given a class with two private data members, a single member function, and a single class member function, the `robcc` transpiler generates a 520 line `.cpp` file cluttered with macros and `#line` directives).

- As a proprietary language used internally, ROB code does not benefit from the C++ tooling ecosystem. Syntax highlighters, static analysis utilities, and automated refactoring tools cannot be used directly with ROB code.

Suffice it to say, many would happily see this infrastructure retired.

Case studies

What follows are a handful of challenges encountered while implementing `rob2`, whose chosen solutions entailed the use of reflection, together with perceived strengths and shortcomings of value-based reflection as implemented by lock3's P2320 compiler.

An index of cases studied is provided for convenience, along with a summary of the reflection features utilized:

- **Inference of entity names**
  Reflection operator and `meta::name_of` from `<experimental/meta>`.

- **Iteration over class members**
  Reflection operator and facilities from `<experimental/meta>`, including: `meta::members_of` and `meta::is_member_function`. Also circumvented an apparent compiler issue using identifier splices.

- **Iteration over transitive base classes**
  Reflection operator, as well as several facilities from `<experimental/meta>`: `meta::base_spec_range`, `meta::type_of`, and `meta::is_class`.

- **Constructing ROB-compatible vtables**
  Reflection operator, identifier splice operator, and many facilities from `<experimental/meta>`, including: `meta::name_of`, `meta::is_public`, `meta::members_of`, `meta::is_static_member_function`, `meta::is_special_member_function`, `meta::is_virtual`, `meta::is_override`, and `meta::is_constructor`.

- **Finding a canonical constructor**
  Reflection operator and several facilities from `<experimental/meta>`, including: `meta::members_of`, `meta::is_member_function`, `meta::is_constructor`, `meta::is_public`, and `meta::is_defaulted`.

- **Overriding default generation of JavaScript bindings**
  `meta::has_attribute`, as well as a modified fork of the P2320 compiler and a custom Clang plugin to circumvent lack of ISO C++ support for user-defined attributes.

Inference of entity names

P2320 value-based reflection makes it exceedingly easy to obtain names from reflections of entities.

```
template <typename CLS>
class RobTypeDescriptorUtil {
    static constexpr const char *k_CLASS_NAME = meta::name_of(^CLS);

    static const RobTypeDescriptor *instance() {
        …
        new … RobTypeDescriptor {
            k_CLASS_NAME,
            …
        };
        …
    }
};
```

Consistently using `meta::name_of` correctly took some practice, since not all reflected entities have names. Although this was never a problem when reflecting on types, it took some time to learn "the right moment" to reflect on certain other entities. For example:

```
template <meta::info INFO>
consteval bool HasName1() { return meta::is_named(INFO); }

template <auto FN>
consteval bool HasName2() { return meta::is_named(^FN); }

struct T { static int mem; };
static_assert(HasName1<^T::mem>());   // Reflection of 'T::mem' has a name.
static_assert(!HasName2<&T::mem>());  // 'Address of T::mem' is not named; name is lost.
```

Although this might be intuitive to those already accustomed to thinking about reflections and ASTs, others might find it surprising that the operations of reflecting on `T::mem` and passing it as a template argument don't commute. Candidly, even after understanding the importance of the order of operations, I still continued to make this mistake at every opportunity.

Takeaways
Getting an entity name mostly "just works," but the behavior can be unintuitive at interface boundaries.

Iteration over class members

Iterating over class members was an essential capability while solving several subproblems, e.g.,

- Reflecting member functions into the virtual table struct.
- Iterating over class data members to ascertain which represent JavaScript properties (i.e., those having some `rob2::ReadProperty<T>`, `rob2::WriteProperty<T>`, or `rob2::ReadWriteProperty<T>` type).
- Iterating over all member functions having some given attribute.

A few points of frustration occurred, but could be circumvented.

- Although `meta::members_of` is advertised to accept arbitrarily many predicate filters P_1, …, P_n, the compiler had difficulty accepting multiple predicates. Most uses of `meta::members_of` were in the context of a `template for` statement, and nesting `if constexpr (`P_2 `&& … &&` P_n`)` inside the loop body generally achieved the same result.

- Once a reflection was obtained, it was sometimes difficult to work with it. For instance, given an iteration over class member functions:

  ```
  template for (constexpr auto e :
                meta::members_of(^CLS, meta::is_member_function))
  ```

  the reflection `e` appears to "forget" the class to which it belongs:

  ```
  static_assert(is_same_v<typename [:meta::type_of(e):],
                          void()>);
  ```

  For some use cases, this is fine; for instance, one can readily invoke the member function on some instance of `CLS`. From this, it is clear the compiler in some way "remembers" that `e` is a member function of `CLS`, despite this information not surfacing through the type.

  ```
  CLS instance;
  instance.[:e:]();
  ```

  On the other hand, I failed to find a means of obtaining a pointer to the member function reflected by `e`. This could be circumvented with an identifier splice (i.e., `&CLS::[#meta::name_of(e)#]`), but this felt like a hack and would, of course, encounter trouble in the presence of overload sets.

These sorts of frustrations were very characteristic of working with the P2320 compiler. Unexpected compile errors, often surfaced as compiler crashes, were quite common. While there is usually a way to work around these issues, it often takes three or four ideas before finding one that the compiler lets through.

Takeaways
`meta::members_of` is one of the most broadly useful facilities from `<experimental/meta>`; it should be easy to use, with as few "sharp corners" as possible. The P2320 compiler's implementation shows some room for improvement.

Iteration over transitive base classes

Generating the virtual table requires iterating not only over direct base classes, but also over all ancestor classes in order from least to most derived. This is implemented as a class template with two public constexpr class member arrays:

```
template <typename CLS>
class BaseClassUtil {
  public:
    static constexpr auto direct_bases = …;  // array<meta::info, M>
    static constexpr auto ancestors    = …;  // array<meta::info, N>
};
```

Three private inner structs describe the "pipeline" of compile-time computations:

1. `struct direct` computes an array of `meta::info`s reflecting the direct base classes.
2. `struct nonunique` computes an array of `meta::info`s reflecting all ancestor classes, with possible duplicates if a common ancestor is shared by multiple parents.
3. `struct unique` computes an array of `meta::info`s reflecting all ancestor classes, with duplicates removed.

Each of these structs implements two consteval class member functions:

- `size_t count()` returns how many base classes belong to the array that will be computed.
- `array<meta::info, …> find()` returns the array.

The toolchain used to build `rob2` lacked support for constexpr `vector`; with the full power of C++20, vectors could be used in lieu of arrays, the `count()` functions could be elided, and the template argument `CLS` representing the type under introspection could be replaced with a `meta::info` function argument. Our implementation, such as it is, can be found here: https://godbolt.org/z/6qsGv6f16.

Takeaways
Iteration over base classes was another essential feature for `rob2`. Value-based reflection facilitates some impressively powerful template-less metaprogramming.

Constructing ROB-compatible vtables

With the help of `BaseClassUtil<CLS>::ancestors`, the virtual table struct (as described by the "ROB virtual tables" section above) can be assembled as a tree of classes inheriting from one another. Care must be taken since, as previously stated, the virtual table not only needs a corresponding field for each member function: it needs each field to *share the same name* as the original member function.

Constructing these virtual tables invariably entails splicing `meta::info` values (e.g., generating wrappers that invoke member functions from JavaScript). Since `meta::info` values received as function arguments cannot be spliced (i.e., because they aren't constexpr), such values will have to be passed as template arguments; we found this property to be "viral", since it forces functions calling these to also receive these `meta::info` values as template arguments. As a result, the code implementing vtable assembly is less "value-based" than much of the rest of `rob2`, leaning more heavily on classic template metaprogramming and fold expressions.

- Define a class template `VtableField<meta::info FN>`, where `FN` is a reflection of a member function. This class has a single non-static data member whose declaration uses the "*identifier splice*" provided by the P2320 compiler (i.e., "`[# "string_constant" #]`") to obtain the same name as the reflected member function:

  ```
  typename c_call_t<FN> [#meta::name_of(FN)#];
  ```

  where if `FN` reflects a function with signature

  ```
  Result (CLS::*)(Arg1, …, Argn)
  ```

  then `c_call_t<FN>` is

  ```
  Result (*)(CLS *, Arg1, …, Argn) .
  ```

- Define a class template
  ```
  template <size_t... Idxs>
  SimpleVtable<typename CLS, index_sequence<Idxs...>>
  : VtableField<findNthVirtualFn(^CLS, Idxs)>... {…}
  ```
  such that the instantiation
  ```
  SimpleVtable<CLS, make_index_sequence<countVirtualFns()>>
  ```
  inherits a non-static function-pointer data member for every virtual member function declared by `CLS`.

- Define `Vtable<CLS>` such that it inherits from `SimpleVtable<BASE>` for every `BASE` in `BaseClassUtil<CLS>::ancestors`.

The `Vtable<CLS>` class has a non-static function-pointer data member for every virtual member function declared by `CLS` or any of its ancestors, and is the required data structure for the vtable.

These data members are filled out by the `Vtable<CLS>` constructor. Iterating over base classes from least to most derived, any definitions for virtual member functions are written to their corresponding fields in the vtable. Iteration order guarantees that the most derived definitions of each function end up in the initialized vtable struct. This vtable calculation utilizes many predicate functions from `<experimental/meta>`: `is_static_member_function`, `is_member_function`, `is_virtual`, `is_override`, `is_public`, `is_special_member_function`, and `is_constructor` were all found to be quite valuable.

Of course, this methodology ignores many possible situations that might arise in C++. For instance, no attempt is made to account for virtual inheritance. Overloading is also not supported, as this would require deeper changes to the system consuming `RobTypeDescriptor`s, as well as implementing some sort of "name mangling" scheme (both of which were outside the scope of this "proof of concept"). However, if such a scheme were pursued, it would likely entail nontrivial string formatting at compile-time. The P2320 compiler's `__concatenate` built-in might facilitate this, but richer compile-time formatting tools (e.g., some constexpr `format`) would be better still.

The use of a hierarchy of templates to build the virtual table circumvents the lack of support for expanding a range of splices into a sequence of class member declarations. Such functionality probably belongs with what P1717 ("*Compile-time Metaprogramming in C++*") calls "source code injection," which deserves more attention once the foundations for reflection and splicing have been standardized. Significantly more elegant generation of these virtual table structs might be made possible by such tools.

Some may read this particular case study and recoil in horror at the "reinvention of the compiler" made possible. I would point out that ROB is hardly unique as a preprocessor language built to compensate for C++'s historical inability to perform such reflections (and for every such preprocessor language that this committee learns of, we can be assured of another `N` about which we hear nothing). While considering what hypothetical new abuses of the language could be pursued with such tools, please also give thought to the decades of macros, hacks, preprocessors, and workarounds that are *already* deployed, which can at last begin their road to retirement if provided with these same tools.

Takeaways

Combining value-based reflection with classic template metaprogramming, together with a handful of facilities from `<experimental/meta>`, makes some truly powerful code introspections now possible. The inability to splice `meta::info` function arguments necessitates the use of templates in places where a more value-oriented style might be otherwise preferred.

Finding a canonical constructor

Due to the aforementioned lack of support for overloading, each `rob2` class must have exactly one "canonical" constructor (i.e., public and user-defined). This constructor is found by iterating over `meta::members_of`, restricted with `meta::is_constructor`, and counting reflections that match both `meta::is_public` and `!meta::is_defaulted`. A compile-time exception is thrown if a unique "canonical" reflection is not found. An implementation can be found here: https://godbolt.org/z/z8sY7v9qs.

A reflection of a canonical constructor for a type `CLS` can be used to instantiate a C-style function template taking the same arguments and returning `CLS *`. A pointer to this wrapper is then stored in the associated vtable struct for `CLS`.

Takeaways
Reflection makes it possible to obtain a "handle" to a constructor in a way that was not before possible (i.e., since one may not obtain a pointer to a constructor). This itself is a very useful operation, and allows for a "canonical constructor" to be found and bound to JavaScript.

Overriding default generation of JavaScript bindings

The ROB language provides several attributes with which a function may be annotated, including a means of suppressing a given function from being surfaced through JavaScript. This feature seemed worth exploring for the `rob2` experiment.

Load-bearing user-defined attributes, however, are currently rendered impossible by [dcl.attr.grammar], which states "*any [...] attribute-token that is not recognized by the implementation is ignored.*" To circumvent this, I modified the lock3 compiler to return the annotation component of Clang's `AnnotationAttr` as its spelling (i.e., instead of the fixed string "`annotate`"). I then developed a simple Clang plugin implementing a `[[rob2::nojs]]` attribute based on `AnnotationAttr`; with this, `rob2` uses `meta::has_attribute` to query for the presence or absence of `[[rob2::nojs]]` on member functions.

```
class RobType : public rob2::CppRobBaseObject
{
    [[rob2::nojs]]
    static void foo();
};
```

The JavaScript `RobType` will have no property named `foo`.

In the context of P2911 ("*Python Bindings with Value-Based Reflection*"), there has been some discussion as to whether attributes are the right mechanism for specifying how language bindings are generated. In particular, the question of who "controls" language bindings has been raised: Is it the job of library authors to decide which APIs can be called from Python, or the right of a library consumer to define that interface?

If `rob2` were an open source framework for generating JavaScript bindings, attributes might be a poor choice of mechanism. One could argue that arbitrary consumers ought to be able to surface classes from third-party C++ libraries through the JavaScript runtime, even if the libraries' authors did not have this use case in mind. This requires that consumers be able to specify bindings separately from the class definition.

In the case of `rob2`, the owners of the JavaScript bindings are also the owners of the C++ implementations. Here, it is explicitly not desirable for library consumers to "reopen" or "extend" the binding surface area of a class. By requiring that bindings be specified within the class definition, control stays with library providers. User-defined attributes are an appealing mechanism for this task.

Proposals for user-defined attributes have already been introduced (e.g., P1887 ("*Reflection on Attributes*") and P2565 ("*Supporting User-Defined Attributes*")), and I add my voice to those supporting such measures.

Takeaways
Once standardized, reflection will find many code generation use cases. When ownership of the generated code resides with the owner of the reflected entity, it makes sense for any "additional data" needed to be attached to declarations. C++ attributes are the perfect mechanism for attaching such data. Relaxing [decl.attr.grammar] to allow user-defined attributes would support such use cases.

Retrospective

The `rob2` user experience

An API was developed for the `rob2` framework that registers a C++ class with the ROB type system by requiring little more than inheritance from a `rob2::CppRobBaseObject` base class.

```
class IAmARobCompatibleClass : public rob2::CppRobBaseObject {
    string d_name;

  public:
    IAmARobCompatibleClass(string_view name)
    : rob2::CppRobBaseObject(rob2::typeof(this))
    , d_name(name) { }

    void greet() {
        cout << "Hello, " << d_name << "!" << endl;
    }
};
…
static const auto types = rob2::RegisterTypes<IAmARobCompatibleClass, OtherClass, …>();
```

With only this, the `IAmARobCompatibleClass` class will be usable from JavaScript.

```
> var obj = new IAmARobCompatibleClass("Dan")
[object …]
> obj.greet()
"Hello, Dan!"
```

There are many satisfying properties of this outcome:

- The need for a proprietary transpiler to C++ is removed.
- The cognitive overhead of juggling a metalanguage on top of C++ is removed.
- The need to generate illegible `.h` and `.cpp` files is removed.
- The "low boilerplate" hallmark of classic ROB code is retained.
- No knowledge of reflection is required to use `rob2`.

Ergonomics of value-based reflection

One of the commonly cited reasons for moving from type-based to value-based reflection is the improved ergonomics, intuition, and "familiarity" of operating on values with functions (e.g., `meta::name_of(^CLS)`) compared to template metaprogramming. Unfortunately since function arguments are never constexpr, any function that splices a `meta::info` argument must receive it as a template argument rather than a function argument, e.g.,

```
template <meta::info Ctor>
auto invokeCtor(auto... args)
```

instead of

```
auto invokeCtor(meta::info Ctor, Ts... args)
```

A function receiving `meta::info`s as template arguments can pass them as function arguments, but the converse does not hold: functions receiving `meta::info`s as function arguments cannot pass them as template arguments. As discussed in the context of constructing `rob2` vtables, the passing of `meta::info`s as template arguments therefore becomes "viral": If some function

```
doComplexTask()
```

instantiates a reflection `meta::info refl`, and through a chain of calls invokes some function

```
doNarrowTask<refl>()
```

which splices `meta::info refl`, all intermediate functions in the call chain between `doComplexTask` and `doNarrowTask` must also receive `meta::info refl` as a template argument. This necessitates a proliferation of templates, which does some harm to the aesthetic otherwise promised by value-based reflection. Revisiting some of the ideas from P1045 ("*`constexpr` Function Parameters*") would go a long way to improve this, even if the resulting "constexpr parameters" were themselves just syntactic sugar over template parameters.

The use of a single monotype `meta::info` also left something to be desired in terms of readability. `rob2` implements utility classes and functions which variously expect a provided `meta::info` to reflect objects as varied as a constructor, a class type, or a non-static member function. With only a single type to represent reflections, it is left to the reader of the code to infer which "sort of object" is expected (and represented) at any given point. One can't help but feel that this sort of "contractual" information belongs as a part of the static type. In the past, SG7 has expressed hesitancy to introduce any sort of "type hierarchy" on top of `meta::info`, partly from a place of concern that it would specify relationships between language entities which today are implicit or unspecified. While I'm sympathetic to such concerns, I still feel that a richer (but small and finite) family of reflection types, in the spirit of what was proposed by P0953 ("*constexpr reflexpr*"), would improve the readability of reflection-heavy code; I won't attempt here to suggest what shape a family of such types might have.

State of the P2320 compiler prototype

The incompleteness of the P2320 compiler is worth noting. Although enough functionality is provided to produce working code for nontrivial use cases, the absence of features like splicing a range of reflections is sorely felt. The inability to do things like obtain pointers-to-member-functions through `meta::members_of` forces the use of odd workarounds, like identifier splices, to accomplish apparently simple objectives. We frequently encountered surprising behaviors, such as:

- `meta::info` values obtained through iteration over `meta::base_spec_range` do not compare equally with themselves (e.g., https://godbolt.org/z/7aKzao1bj).
- Identifier splices can be used to declare entities having malformed names (e.g., https://godbolt.org/z/fn8Kb9GzW).

The P2320 compiler has been an extremely valuable tool for experimentation, but it would be reassuring to see a more complete and well-behaving demonstration of value-based reflection before advancing any proposal towards standardization.

Conclusions

Value-based reflection is sufficiently powerful to replace a language like ROB with pure C++. It can automate the generation of JavaScript bindings from a class, even when constrained to target a less than ideal legacy interface rooted in C. I come away from the `rob2` experiment enthusiastic and optimistic for the future of value-based reflection.

Many of the features proposed for value-based reflection were essential to `rob2`'s implementation. The reflection, splice, typename splice, and identifier splice operators were all heavily used, along with many library functions from `<experimental/meta>`. Support for something equivalent to the proposed "`...[:range:]...`" syntax for splicing a range of reflections was sorely missed, and seems like a must-have for any final proposal.

In the case of `rob2`, *user-defined attributes* proved essential. They enabled the attachment of additional data to declaration sites, which yielded a very natural syntax for customizing code generation.

While promising, the existing value-based syntax shows some shortcomings in projects of reasonable size. The use of a single `meta::info` type becomes confusing in the presence of many reflections representing a diverse set of entities. Which kind of entity a reflection "represents" must be inferred from context clues, but it feels like this ought to belong to the type. Furthermore, the aesthetic appeal of representing reflections as values is lessened by the frequent need to pass them as template arguments. "Constexpr parameters", or something like them, would improve the situation.

The lock3 compiler is an amazing tool for experimentation, but it would be helpful to see a more complete and stable demonstration of this technology before standardizing value-based reflection.

Acknowledgments

Sincere thanks to everybody in the C++ community who has contributed to the search for the right model for reflection – it's been a long road, but it's getting somewhere exciting. Special thanks to the lock3 folks for producing and sharing the P2320 compiler, without which I never would have started this experiment. Thanks also to Jagrut Dave, Joshua Berne, and Sergei Murzin, who provided valuable feedback on this paper, and to Mark Sciabica, who identified opportunities to simplify aspects of the `rob2` implementation.