Relaxing the One Definition Rule in Interpreted C++

Javier López-Gómez
Department of Computer Science
University Carlos III of Madrid
28911–Leganés, Spain
jalopezg@inf.uc3m.es

Javier Fernández
Department of Computer Science
University Carlos III of Madrid
28911–Leganés, Spain
jfmunoz@inf.uc3m.es

David del Rio Astorga
Department of Computer Science
University Carlos III of Madrid
28911–Leganés, Spain
drio@pa.uc3m.es

Vassil Vassilev
Princeton University
New Jersey 08544, United States
vvasilev@cern.ch

Axel Naumann
Experimental Physics
CERN
1211 Geneva 23, Switzerland
axel.naumann@cern.ch

J. Daniel García
Department of Computer Science
University Carlos III of Madrid
28911–Leganés, Spain
jdgarcia@inf.uc3m.es

Abstract

Most implementations of the C++ programming language generate binary executable code. However, interpreted execution of C++ sources has its own use cases as the Cling interpreter from CERN's ROOT project has shown. Some limitations are derived from the ODR (One Definition Rule) that rules out multiple definitions of entities within a single translation unit (TU). ODR is there to ensure uniform view of a given C++ entity across translation units. Ensuring uniform view of C++ entities helps when producing ABI compatible binaries. Interpreting C++ presumes a single ever-growing translation unit that define away some of the ODR use-cases. Therefore, it may well be desirable to relax the ODR and, consequently, to support the ability of developers to override any existing definition for a given declaration. This approach is especially well-suited for iterative prototyping. In this paper, we extend Cling, a Clang/LLVM-based C++ interpreter, to enable redefinitions of C++ entities at the prompt. To achieve this, top-level declarations are nested into inline namespaces and the translation unit lookup table is adjusted to invalidate previous definitions that would otherwise result in ambiguities. Formally, this technique refactors the code to an equivalent that does not violate the ODR, as each definition is nested in a different namespace. Furthermore, any previous definition that has been shadowed is still accessible by means of its fully-qualified name. A prototype implementation of the presented technique has been integrated into the Cling C++ interpreter, showing that our technique is feasible and usable.

CCS Concepts • **Software and its engineering → Interpreters**; **Just-in-time compilers**; **Compilers**; **General programming languages**; **Language features**; **Rapid application development**; *Scripting languages.*

Keywords C++, interpreter, One-Definition-Rule, Cling

ACM Reference Format:
Javier López-Gómez, Javier Fernández, David del Rio Astorga, Vassil Vassilev, Axel Naumann, and J. Daniel García. 2020. Relaxing the One Definition Rule in Interpreted C++. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377555.3377901

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377901

1 Introduction

Recently, interpreted languages have been widely adopted for application prototyping in multiple areas and to aid unexperienced users in defining the logic of their applications. In that regard, application developments based on compiled languages for performance issues can benefit of using an interpreter of the same language for rapid prototyping in order to reduce the time-to-market. Following this idea, the CERN's ROOT project has demonstrated that the use of a C++ interpreter (Cling) can reduce the necessary effort for developing prototypes and transforming them into high-performance applications.

However, since the C++ language has been designed to be compiled, interpreting this language in a user-friendly way presents some challenges. In this paper, we focus on providing Cling with the functionality of redefining entities, such as variables, functions, and types, in a similar way to other interpreted languages like Python. To do so, it is necessary to relax the C++ One Definition Rule so that we allow more than one definition per translation unit.

In this paper, we present a formalization for supporting entity redefinition on interpreted C++ and we implement this

behavior on Cling as a validation of the proposed technique. Specifically, this paper contributes with the following:

- We present a formalization for relaxing the ODR in C++ that can be leveraged on any C++ interpreter.
- We implement an Abstract Syntax Tree (AST) transformer to support entity redefinition on a real-world C++ interpreter.
- We analyze the output of the new transformer to validate the proposed technique, and evaluate the possible overhead.

The rest of this document is organized as follows. Section 2 revisits some related works in the area. Section 3 gives an overview of the CERN's ROOT framework and the Cling interpreter. Section 4 presents the formalization for supporting entity redefinition. Section 5 describes the implementation of the required AST transformer for relaxing the ODR in Cling. In Section 6, we employ some examples to validate our proposal and evaluate the overhead introduced by the required additional handling. Finally, Section 7 closes this paper with some concluding remarks and future works.

2 State of the Art

Interpreted languages have become popular in the industrial and scientific areas. This is mainly due to three important characteristics: *(i)* the adoption of agile software development methodologies based on fast application prototyping (Rapid Application Development [16]); *(ii)* the need to provide tools that ease the application development for non-experts, which is important in scientific areas and industrial data management; *(iii)* the increased portability with respect to compiled languages. For instance, Scala [18] has been widely adopted for managing large data sets in Big Data applications. On the other hand, Python [20] has become even more popular thanks to its high-level abstractions that help domain experts to develop scientific applications [19].

However, interpreted languages are slower than compiled ones due to the instruction generation at run-time and the difficulties for exploiting the available resources in a given platform. Thus, to increase the performance, interpreters of these languages leverage Just-in-Time (JIT) compilation techniques to generate a compiled version of application hot-paths, e.g. PyPy [21], HOPE [2]. Additionally, multiple libraries implemented in high-performance compiled languages provide bindings to be used on interpreted applications to improve the performance, e.g. TensorFlow [1]. Moreover, it is worth mentioning that almost every contemporary programming language has a Read-Eval-Print-Loop (REPL) also known as language shell, e.g. Swift[4], that despite being a compiled language heavily supports REPL-style development.

Nevertheless, in order to obtain the maximum performance and to minimize response times in a production build of the application, it is necessary to generate a compiled binary or to transform the interpreted application to compiled languages. In this sense, we can find two major approaches: *(i)* tools that allow generating compiled applications or bytecode that runs on a Virtual Machine (VM) from an interpreted language script [17], and *(ii)* the development of interpreters for typically compiled languages to reduce the code transformation for the production version.

Some examples of tools able to compile Python scripts are Cython [5], Jython [12] and IronPython [9]. For instance, Cython is a Python and C compiler that can generate optimized binaries. However, this tool requires to use a superset of the Python language to exploit the available resources such as the general lock releasing for exploiting thread parallelism. For this reason, these tools are mainly used for implementing libraries that will be used from an interpreted script.

On the other hand, several interpreters of C and C++ languages can be found: Ch [6], Clip [15], CInt [10], UnderC [8] and Cling [22]. These tools allow developers to take advantage of interpreted languages for fast application prototyping having, as a result, a code that can be compiled with minimal efforts. Compiled language standards and, C++ specifically, presents some limitations to be used as an interpreted language. An example of these limitations is the C++ ODR that avoids entity redefinition in the same translation unit [11]. This limitation is not required in interpreted languages since the definition of an entity depends only on the interpretation order of the script code. This way an entity definition is valid until its next redefinition. In this paper, we present a technique to allow entity redefinition on a C++ interpreter while keeping the C++ language consistent.

3 Background

In this section, we describe the ROOT project and its C++ interpreter (Cling), widely used in the community of High-Energy-Physics (HEP) and other scientific areas.

3.1 ROOT Project

ROOT[3] is a cross-platform C++ framework for data processing in the high-energy physics area, developed mostly at CERN. This framework is designed for storing and analyzing large amounts of data. Basically, it provides the following components:

Data model. The ROOT framework provides a data model that allows to store data, represented as C++ objects, into compressed binary machine-independent files. Those binary files also store the format description of the data, allowing access to the information from anywhere.

Statistics and data analysis libraries. ROOT also provides a huge set of tools for mathematical and statistical analysis that can easily operate over ROOT

files. Furthermore, it also provides visualization tools to display histograms, scatter plots and function fitting. Additionally, these tools take full advantage of C++ features and parallel processing techniques.

Interactive C++ interpreter. This component provides a C++ interpreter (Cling) for interactive developing and to compile the resulting application to exploit the available resources. This interpreter can also be used with user-friendly development environments designed for interpreted languages such as Jupyter notebook, either through ROOT or the Xeus-cling [7] project.

Other language bindings. ROOT provides a set of bindings that allow to use the framework with different languages such as Python, R and Mathematica. In the context of this paper, the Python bindings (PyROOT/cppyy [14]) are especially relevant as they leverage Cling to access the C++ side at run-time.

3.2 Cling

Cling is a Clang/LLVM-based C++ interpreter developed at CERN, that has been adopted as the interpreter for the ROOT project. Cling leverages the Clang/LLVM infrastructure for parsing and code generation, meaning that it only has to deal with issues derived of C++ interpretation. This keeps Cling codebase reasonably small (about 36K LOC) and eases maintenance. An overview of Cling is shown in Figure 1.

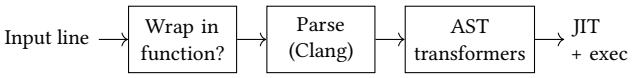

Figure 1. Cling input transformation

In general, Cling users expect a Python-like interaction. In other words, the user expects the interpreter to accept an statement, even if it does not appear as a part of the body of a function. However, this practice usually results in ill-formed code according to ISO C++[11]. If the input line cannot be proved to be valid, it will be wrapped in a uniquely-named function. At this stage, several simple cases can be detected as valid (functions, classes, namespaces, etc.). However, Cling is not able to do so for variable declaration, such as "`int i = 0;`". This is fixed lated by the DeclExtractor transformer, which extracts declarations out of the wrapper functions. Cling also supports a "raw input" mode, in which this "wrapping in functions" stage is skipped completely.

After turning the user code into valid C++, it can be normally parsed by Clang. The output of this stage is the abstract syntax tree (AST) for the parsed top-level declarations. Clang also adds these to the translation unit declaration list. In this sense, the TU is constructed incrementally.

The generated AST for top-level declarations, i.e. those that appear at the TU level, may be transformed to support other Cling features. This processing is performed by independent transformation blocks which are executed sequentially after the AST is created. The former blocks may be classified as either an ASTTransformer (apply to all parsed declarations), or WrapperTransformer (apply only to wrappers generated in the first stage). For example, declaration statements that were previously wrapped into a function must be moved back to the global scope (TU), which is done by the DeclExtractor transformer. Figure 2 shows the modifications performed by DeclExtractor for the input line "`int i = 0, j;`". Additionally, Cling includes transformers to support other features, e.g. `auto` specifier synthesis, invalid memory reference protection, etc.

```
|-●◂
`-FunctionDecl __cling_Un1Qu30 'void (void *)'
  |-ParmVarDecl vpClingValue 'void *'
  `-CompoundStmt
    |-DeclStmt
    | |-VarDecl i 'int' cinit
    | | `-IntegerLiteral 'int' 0
    | `-VarDecl j 'int'
```

Figure 2. Transformation performed by DeclExtractor

The last step in the interpreter pipeline is just-in-time (JIT) compilation and execution. Cling offloads this task on LLVM.

4 Proposal for Entity Redefinition

This section introduces the proposed technique to override a previous definition for a given declaration. The described procedure relies on nesting each redeclaration into its own scope by using C++ inline named namespaces. Therefore, using this technique does not incur in a violation of the ODR, nor requires major changes to the compiler. According to ISO C++[11], members of an inline namespace can be accessed as if they are members of the enclosing namespace, i.e. names introduced by such namespace "leak" to the enclosing scope. However, as shown in Listing 1, if a name is made available in the enclosing scope through more than one inline namespace, unqualified lookup for the given name is ambiguous. In Section 4.3, we tackle this issue by manually adjusting the lookup table of the enclosing scope.

In addition to the technique described in this section, the following approaches were analyzed and finally discarded in favour of the current proposal: *(i)* instead of manually adjusting the TU lookup table, we considered removing the previous declaration from the AST and the matching JIT'ed symbol, but this causes problems with type/variable definitions, as they may be still in use by user code, e.g. redefining a type while there is a variable in scope that still uses the old type definition, or redefining a variable while being used by a thread; and *(ii)* in order to keep C++ more conformant, we considered to replace the changes in the TU lookup table by another AST transformation for each reference to a

namespaced entity, so that it is qualified with the name of the latest generated namespace; however, this is not feasible because most compilers (including Clang) only have the AST available after semantic analysis, which cannot be passed by C++ code that contains ambiguous names.

```
inline namespace ns0 { int i = 0; }
inline namespace ns1 { double i = 1.0; }
auto j = i; //unqualified lookup is
    ambiguous
```

Listing 1. Ambiguous unqualified lookup

4.1 Covered Cases and Exceptions

Not all declarations that introduce a name are subject to the aforementioned transformation. Instead, it can only be applied in contexts where an inline namespace may be used. Therefore, to avoid ill-formed namespace constructs, we restrict this transformation to the translation-unit level. Similarly, only named declarations that are definitions, or that may be defined later, i.e. forward declarations, should be moved into a namespace.

Additionally, some declarations that introduce a name must not be nested into a namespace, either because repetition is allowed, or because nesting them changes the original meaning. This includes:

using-directive e.g. `using namespace std`, that makes all the names in `std` visible for unqualified name lookup. In this case, such declarations shall not be transformed, since issuing twice a using-directive in a given scope does not pose any problem.

using-declaration e.g. `using std::vector`, that makes `vector` accessible for unqualified lookup in the current scope. The same rationale applies for these declarations.

4.2 Rules for AST Transformation

The proposed transformation may be formally described using a syntax-directed definition (SDD)[13], that appends semantic rules to the grammar productions relating to declarations whose redefinition is to be allowed. In this notation, each grammar production has been associated a set of semantic rules that are evaluated in the specified order. Each rule may either set the value of an attribute for the given entity, e.g. `E.attr = . . .`, or call a function that may have side-effects.

As shown in Table 1, for top-level redefinable declarations, the semantic declaration context (`DeclContext` attribute) is set to a synthesized uniquely-named inline namespace, which in turn is added to the translation unit. Added semantic rules has been typesetted in bold face. Due to space limitation, only some productions are shown.

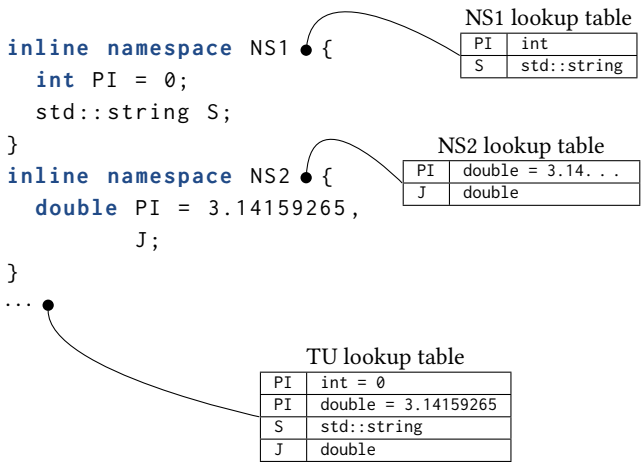

Figure 3. Lookup tables for translation unit and inline namespaces

4.3 Invalidation of Ambiguous Unqualified Names

Rules in Table 1 cause target top-level declarations to be nested into inline namespaces. Because inline namespaces make their members visible in the enclosing scope, declared names may still be accessed as if they were not part of a namespace. However, if the same name is "leaked" via different namespaces, unqualified lookup will fail due to ambiguity. Instead, such lookups should resolve to the latest declaration.

4.3.1 Removing Ambiguity

To that aim, ambiguous lookups must turn into non-ambiguous that return the expected result. As shown in Figure 3, each declared name in the inline namespace (NS1) is made visible not only in the namespace lookup table, but also in that of the enclosing scope (TU). If the same name is made visible by several namespaces (NS1, NS2, ...), then there will be more than one entry with the given name in the TU lookup table.

Therefore, to get rid of ambiguity, existing entries for the given name must be removed, provided that they cannot be considered an overload. An overload is a set of declarations that despite having the same name, the compiler is able to disambiguate using the number and/or type of the arguments. This adjustment is made by the `fix_TU_lookup_table()` function. As can be seen in Algorithm 1, this function looks up the specified name in the TU scope and iterates through the results invalidating ambiguous names that introduce a previous definition (lines 4—18). The `InInlineNS()` function returns whether the given declaration is part of a synthesized inline namespace. Note that, enumerators introduced by an unscoped enumeration are reachable from the TU scope through unqualified lookup and should be invalidated. Also, further non-definition redeclarations of the same entity are discarded (lines 20—22) to avoid ambiguity between the last

Table 1. Syntax-directed definition to nest declarations into a namespace

Production	Semantic Rules
function-definition$_D$ → ··· declarator virt-specifier-seq$_{opt}$ function-body	D.node = new FunctionDefinition(. . . , declarator, function_body) D.DeclContext = new Namespace("__NS_xxx", INLINE, { FD.node }) D.node = FD.DeclContext
simple-declaration$_D$ → ··· decl-specifier-seq init-declarator-list ';'	D.node = new SimpleDeclaration(. . .) D.DeclContext = new Namespace("__NS_xxx", INLINE, { D.node }) D.node = FD.DeclContext
⋮	⋮

definition and the new non-definition declaration that is part of a different namespace.

Moreover, if the whole functionality is encapsulated in the `allow_redefine()` function shown in Listing 2, then allowing a declaration to adopt a new definition may be accomplished only by adding a call to `allow_redefine()`, as shown in the syntax-directed translation scheme (SDT)[13] excerpt in Listing 3.

```
D.DeclContext = new Namespace("__NS_xxx",
    INLINE, { D.node })
D.node = D.DeclContext
fix_TU_lookup_table(D)
```

Listing 2. The `allow_redefine()` function used in the SDT

4.3.2 Exceptions: Overloads, Unscoped Enumerations, etc.

Some particular cases require either to preserve existing lookup table entries, or to invalidate additional ones, namely:

Function overloads. If all the duplicated entries refer to a function overload, none of them shall be removed. In this case, the lookup result is said to be overloaded (not ambiguous). Additionally, ISO C++ paragraph [temp.over.link]p4[11] must be verified for overloaded templated functions.

Unscoped enumerations. An unscoped enumeration is a transparent context, i.e. enumerators are made visible in the parent context. Because declared enumerators are made visible in the enclosing inline namespace, and therefore in the translation unit, the removal of all those names from the TU lookup table shall also be considered.

Declaration after definition. Any non-definition declaration that comes after a definition is ignored, e.g.

```
class C { ... };
class C; // ignored
```

5 Cling Implementation

This section describes the implementation of the aforementioned technique on top of the Cling C++ interpreter. Given

```
1  Function fix_TU_lookup_table(D)
   Data: D: new declaration that introduces a name
2  begin
3      Previous ⟵ LookupTUName(D);
4      for p ∈ Previous do
5          if IsDefinition(p) ∧
            (¬ IsDefinition(D) ∨ ¬ InInlineNS(p)) then
6              continue;
7          end
8          if p, D are function declarations/templates
            ∧ IsOverload(D, p) then
9              continue;
10         end
11         RemoveFromTULookupTable(p);
12         if p is an unscoped enum then
13             E ⟵ EnumeratorsOf(p);
14             for e ∈ E do
15                 RemoveFromTULookupTable(e);
16             end
17         end
18     end
19     // Ignore further non-definition redeclarations
20     if |Previous| ≠ 0 ∧ ¬IsDefinition(D) then
21         DiscardDeclaration(D);
22     else
23         AddDeclaration(D);
24     end
25 end
```

Algorithm 1: The `fix_TU_lookup_table` function

that Cling's architecture allows for AST transformation before the JIT compilation takes place, all the additional handling required for supporting redefinition has been fitted in the new DefinitionShadower AST transformer[1].

Cling AST transformers run in strict order in which they are registered. As will be discussed in Sections 5.1.2 and 5.1.3,

[1] DefinitionShadower has been merged into Cling master branch. See https://github.com/root-project/cling/.

```
function-definition → attribute-specifier-seq_opt decl-specifier-seq_opt declarator virt-specifier-seq_opt
                      function-body  { ···;
                                       allow_redefine(); }
simple-declaration → decl-specifier-seq init-declarator-list_opt ';'
                      | attribute-specifier-seq decl-specifier-seq init-declarator-list ';'
                                     { ···;
                                       allow_redefine(); }
...
```

Listing 3. Modified SDT that allows redefining entities

DefinitionShadower must run before the existing DeclExtractor transformer to produce the expected behavior.

5.1 The DefinitionShadower AST Transformer

This transformer employs the aforementioned "shadowing" technique, and therefore requires to rewrite most top-level declarations as if they were nested into an `inline namespace`, and to apply the fixes detailed in Section 4.3 to the lookup table of the enclosing scope (TU), so that unqualified lookup always resolves to the latest declaration. These changes do not require a patch to Clang sources, and can be entirely implemented in Cling.

5.1.1 Namespacing Top-Level Declarations

The `DefinitionShadower::Transform(Decl *)` function implements the transformation described in Section 4.2. Specifically, it performs the following: *(i)* creating –if needed– a uniquely-named per-transaction NamespaceDecl node (referred to as `DefinitionShadowNS`) that has been marked as `inline`, and adding it to the TranslationUnitDecl declaration list; *(ii)* removing the given named declaration from the TranslationUnitDecl declaration list; *(iii)* setting its declaration context to the `DefinitionShadowNS` namespace; and *(iv)* adding it to the `DefinitionShadowNS` declaration list.

Note that, step *(iii)* fails for out-of-line member function definitions, because the semantic declaration context should be the CXXRecordDecl of the class, and cannot be changed. Therefore, out-of-line member functions cannot be directly shadowed. As a workaround, the owning class has to be redefined prior to attaching a new out-of-line function definition.

Additionally, because function template instantiations inherit the declaration context of the templated declaration, the instantiation pattern must also be updated. Otherwise, if we try to redefine a templated function, the mangled name for template instantiations may clash with a previous definition of the same template.

5.1.2 Adjusting the Translation Unit Lookup Table

The required patching to the translation-unit lookup table is performed by the `invalidatePreviousDefinitions(Decl *D)` function. Provided that `D` is a definition, this function hides from Sema lookup any previous definition of the same entity. Note that, while unqualified lookup will only return the latest definition, it still allows reachability of shadowed declarations via qualified lookup, e.g. `__cling_N50::decl`.

The previous function checks whether the given declaration is a wrapper function generated by Cling, in which case we iterate through all local declarations (that will be moved by DeclExtractor), invalidating any previous global definition.

`invalidatePreviousDefinitions(NamedDecl *D)` handles the invalidation of any previous definition of a named declaration. In general, we lookup the given name in the translation unit and iterate through the results, skipping over non-definitions. Candidates for removal are checked for function/template overload using the `Sema::IsOverload()` function, and if so they are kept. Otherwise, we remove the declaration from the StoredDeclsList (lookup table) of the translation-unit. As an special case, because unscoped enumerations "leak" enumerator names to the enclosing scope, we also invalidate any previous definition of the enumerators.

Also, because some Cling extensions cache information about declarations, e.g. TCling, we registered an interpreter callback that provides notification when a definition has been shadowed. Therefore, the new `DefinitionShadowed` callback may be used in that case to erase cached information.

5.1.3 Modifications to Declaration Extraction

The implementation required minor changes to the DeclExtractor transformer, so as to properly move declarations to the enclosing scope. The unmodified DeclExtractor incorrectly assumed that this scope is always the translation unit. However, if DefinitionShadower is enabled, the wrapper function has been moved to an `inline namespace` and declarations should be extracted onto it, as can be seen in Figure 4.

This fact also implies that the DefinitionShadower transformer should always run before DeclExtractor.

5.2 Enabling/Disabling the New Transformation

If registered, the AST transformer may be turned on/off for the next input line by means of the `EnableShadowing` compilation option. Compilation options control several aspects

```
`-NamespaceDecl __cling_N50 inline
  |-●◂
  `-FunctionDecl __cling_Un1Qu30 'void (void *)'
    |-ParmVarDecl vpClingValue 'void *'
    `-CompoundStmt
      |-DeclStmt
      | `-VarDecl i 'int' cinit
      |   `-IntegerLiteral 'int' 0
```

Figure 4. New DeclExtractor behavior

of Cling, such as the optimization level or toggling a feature, e.g. declaration extraction, invalid memory reference protection, etc.

`EnableShadowing` is set to 0 if Cling raw input is enabled. Otherwise, if `EnableShadowing` equals 1, valid named top-level declarations shall be transformed except in the following cases:

Not typed in the Cling prompt. Shadowing is enabled only for declarations that were parsed from an input line, therefore disabling it for `#include`'ed files; otherwise, it might break system header files. Because Cling stores input lines in a virtual file with overriden contents, they may be easily recognized based on their source location.

Is a UsingDirectiveDecl/UsingDecl. As discussed in Section 4.1, `using-directive` and `using-declaration` should not be transformed.

Is a NamespaceDecl. Shadowing namespace members is currently not supported.

Is a function template instantiation. Cling copies input lines in a distinct virtual file and starts parsing it. Consequently, at end of file, `ASTConsumer::HandleTranslationUnit()` emits pending template instantiations. These instantiations are fed through AST transformes as top-level declarations, and should be ignored by DefinitionShadower.

5.3 Other Minor Changes

Cling is able to pretty-print the type and value of an expression. This behavior is automatically turned on if an input line is not terminated by a semicolon. However, nesting type declarations into a namespace changes the qualified name of the type, which affects how it is printed.

Because the proposed transformation moves most top-level `NamedDecl` nodes into a namespace, their fully qualified name changes w.r.t. the original typename as seen by the user. Take the input "`class MyClass { . . . } X`" as an example. As shown in Figure 5.b, `X` is pretty-printed by Cling as "`(class __cling_N50::MyClass &) @0x7f0. . .`" after enabling DefinitionShadower.

As can be seen, the typename shown in the output changes w.r.t. Figure 5.a. The issue is fixed by setting the PrintingPolicy flag `SuppressUnwrittenScope = 1` in `ValuePrinter.cpp`. This flag specifies whether to print parts of qualified names that are not required to be written, e.g. inline/anonymous namespaces.

(a) Original Cling ValuePrinter output

```
root [0] class MyClass {} X
(class MyClass &) @0x7fb4deb45008
```

(b) DefinitionShadower enabled

```
root [0] class MyClass {} X
(class __cling_N50::MyClass &)
    @0x7f0f2da63008
```

(c) DefinitionShadower enabled and fixed ValuePrinter

```
root [0] class MyClass {} X
(class MyClass &) @0x7fdf6baac008
```

Figure 5. Cling pretty-print for "`class MyClass {} X`"

5.4 Limitations

While the current implementation closes the behavioral gap between the Cling C++ interpreter and other interpreted languages, e.g. Python, it has some known limitations that restrict its use, namely:

Shadowing a global object does not free storage. In C++, an *l-value* is an object that has a memory location, e.g. a variable, and therefore it may appear on the left-hand-side of an assignment expression. A shadowed *l-value* cannot be found via unqualified lookup, but the memory it was referring to is still allocated. Furthermore, these objects can be referenced using their qualified name.

Changes in RTTI type information. Run-Time Type Identification (RTTI) is a C++ mechanism for type introspection. As discussed in the previous section, nesting type declarations into a namespace changes the qualified name of types, which might be a problem for applications heavily relying on RTTI. Fixing this issue requires additional patches to the compiler.

6 Validation

This section presents an analysis of Cling DefinitionShadower behaviour to validate the correctness of the proposed entity redefinition technique. To do so, we provide a close-up view of the resulting AST and lookup table state for a set of examples that covers most of the recurrent uses of interpreted C++.

Table 2 shows the step-by-step sequential execution of interpreted code in Cling, including the transformed AST and the lookup table state.

Table 2. Step-by-step execution of interpreted code

Code

```
1 int i = 1;
```

Transformed AST

```
|-NamespaceDecl __cling_N50 inline
| |-VarDecl used i 'int' cinit
| | `-IntegerLiteral 'int' 1
| `-FunctionDecl __cling_Un1Qu30 'void (void *)'
```

TU lookup table

Name	Type	Value
i	int	1

Code

```
2 double i = 3.141592;
```

Transformed AST

```
|-NamespaceDecl __cling_N51 inline
| |-VarDecl used i 'double' cinit
| | `-FloatingLiteral 'double' 3.141592e+00
| `-FunctionDecl __cling_Un1Qu31 'void (void *)'
```

TU lookup table

Name	Type	Value
~~i~~	~~int~~	~~1~~
i	double	3.141592

Code

```
3 char f(int x) { return 'X'; }
4 int f() { return 0; }
```

Transformed AST

```
|-NamespaceDecl __cling_N52 inline
| `-FunctionDecl f 'char (int)'
|   |-ParmVarDecl x 'int'
|   `-CompoundStmt
|     `-ReturnStmt
|       `-CharacterLiteral 'char' 88
|-NamespaceDecl __cling_N53 inline
| `-FunctionDecl f 'int (void)'
|   `-CompoundStmt
|     `-ReturnStmt
|       `-IntegerLiteral 'int' 0
```

TU lookup table

Name	Type	Value
i	double	3.141592
f	char (int)	
f	int ()	

Code

```
5 double f() { return 1.0; }
```

Transformed AST

```
|-NamespaceDecl __cling_N54 inline
| `-FunctionDecl f 'double (void)'
|   `-CompoundStmt
|     `-ReturnStmt
|       `-FloatingLiteral 'double' 1.000000e+00
```

TU lookup table

Name	Type	Value
i	double	3.141592
f	char (int)	
~~f~~	~~int ()~~	
f	double ()	

Code

```
6 template <typename T>
7 struct S { T i; };
8
9 S<int> f{99};
```

Transformed AST

```
|-NamespaceDecl __cling_N56 inline
| `-ClassTemplateDecl S
|   |-TemplateTypeParmDecl typename depth 0 index 0 T
|   |-CXXRecordDecl struct S definition
|   | `-FieldDecl i 'T'
|   `-ClassTemplateSpecializationDecl struct S
|     |-TemplateArgument type 'int'
|     `-FieldDecl i 'int':'int'
|-NamespaceDecl __cling_N57 inline
| |-VarDecl f 'S<int>':'__cling_N56::S<int>'
| | `-InitListExpr 'S<int>':'__cling_N56::S<int>'
| |   `-IntegerLiteral 'int' 99
| `-FunctionDecl __cling_Un1Qu33 'void (void *)'
```

TU lookup table

Name	Type	Value
i	double	3.141592
~~f~~	~~char (int)~~	
~~f~~	~~double ()~~	
S<T>	__cling_N55::S<T>	
f	__cling_N55::S<int>	{99}

Code

```
10 template <typename T>
11 struct S { T i, j; };
12
13 S<double> g{0, 33.0}
```

Transformed AST

```
|-NamespaceDecl __cling_N58 inline
| `-ClassTemplateDecl S
|   |-TemplateTypeParmDecl typename depth 0 index 0 T
|   |-CXXRecordDecl struct S definition
|   | |-FieldDecl i 'T'
|   | `-FieldDecl j 'T'
|   `-ClassTemplateSpecializationDecl struct S
|     |-TemplateArgument type 'double'
|     |-FieldDecl i 'double':'double'
|     `-FieldDecl j 'double':'double'
|-NamespaceDecl __cling_N59 inline
| |-VarDecl f 'S<double>':'__cling_N58::S<double>'
| | `-InitListExpr 'S<double>':'__cling_N58::S<double>'
| |   |-FloatingLiteral 'double' 0.000000e+00
| |   `-FloatingLiteral 'double' 3.300000e+01
| `-FunctionDecl __cling_Un1Qu34 'void (void *)'
```

TU lookup table

Name	Type	Value
i	double	3.141592
~~S<T>~~	~~__cling_N55::S<T>~~	
f	__cling_N55::S<int>	{99}
S<T>	__cling_N57::S<T>	
g	__cling_N57::S<int>	{0,33.0}

Code

```
14 using namespace std;
15 namespace NS {
16   string s("Cling");
17 }
```

Transformed AST

```
|-UsingDirectiveDecl Namespace 'std'
|-NamespaceDecl NS
| `-VarDecl s 'std::string':'std::basic_string<char>'
|   `-ExprWithCleanups
|     `-CXXConstructExpr
|       |-ImplicitCastExpr 'const char *'
|       | `-StringLiteral 'const char [6]' lvalue "Cling"
|       `-CXXDefaultArgExpr
```

TU lookup table

Name	Type	Value
i	double	3.141592
f	__cling_N55::S<int>	{99}
S<T>	__cling_N57::S<T>	
g	__cling_N57::S<int>	{0,33.0}
NS	[namespace]	

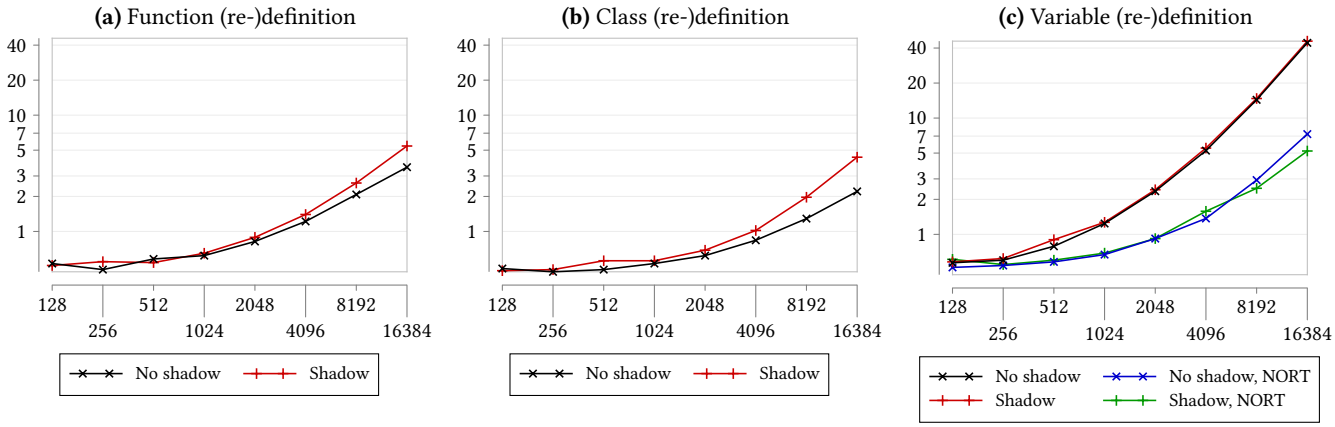

Figure 6. Cling run time plots (both shadowing enabled/disabled)

In the first line, an integer variable (`int i`) is declared. This declaration is wrapped in a function named `__cling_Un1Qu-30`. Then the AST tree is generated, and both DefinitionShadower and DeclExtractor transformers are executed. First, DefinitionShadower transforms the AST by nesting the function into the `__cling_N50` inline namespace. Then, DeclExtractor extracts the declaration out of the wrapper function, yielding the AST shown in Table 2. Given that this is the first declaration named `i`, the TU lookup table is not modified.

In the second line, a variable with the same name but different type (`double i`) is declared. As before, the declaration is wrapped in a function named `__cling_Un1Qu31`. After the AST is created, DefinitionShadower nests the function into the `__cling_N51` inline namespace, and also removes the previous entry for the given named declaration from the TU lookup table. Finally, DeclExtractor extracts the declaration out of the wrapper function.

Lines three and four declare two different functions with the same name. However having both of them different parameters, it can be considered a function overload. These input lines do not require to be wrapped. However, both are nested into inline namespaces (`__cling_N52` and `__cling_-N53`, respectively) by DefinitionShadower. In this case, DeclExtractor does not do anything and the TU lookup table is not modified.

Line five declares a function with the same name and parameters as the one on line four. As before, this declaration does not require a wrapper. Again, DefinitionShadower nests the declaration into namespace `__cling_N54`. This transformer also removes the previous entry with the same name from the TU lookup table.

Lines six through nine declare a templated structure with one member (`struct S`), and an instance of `S<int>` with the same name as the functions presented on lines three, four and five. In this case, only the variable declaration has been wrapped into function `__cling_Un1Qu30`. DefinitionShadower nests the templated structure, along with its specializations, into an inline namespace (`__cling_N56`). The function wrapping the variable declaration is nested into a different inline namespace (`__cling_N57`). This transformer also removes all previous entries on the TU lookup table that have the given name. Finally, DeclExtractor extracts the declaration out of the wrapper function.

Lines ten through thirteen replace the templated structure introduced in lines six and seven. Also, we declare an instance of this structure (`S<double> g`). As before, the variable declaration requires the wrapper function `__cling_Un-1Qu34`. DefinitionShadower nests the templated structure, along with its specializations, into the `__cling_N58` namespace. On the other hand, the variable declaration is nested into a different inline namespace (`__cling_N59`). The transformer also removes the entry for the previous declaration of the structure from the TU lookup table. Finally, DeclExtractor extracts the variable declaration out of the wrapper function.

Finally, in lines fourteen through seventeen, we introduce a using directive (`using namespace std`) and the `NS` namespace. Neither of these declarations have to be wrapped into a function. Also, DefinitionShadower does not modify the AST because both, using directives and user-defined namespaces are considered exceptions. In this case, DeclExtractor does not do anything and the TU lookup table is not modified.

As can be seen, this proposal improves the user experience of using interpreted C++ for fast application prototyping, while the code can still be reused for the high-performance compiled version. Moreover, in a Jupyter notebook environment the user is allowed to edit existing cells and change type/function definitions.

Finally, for the sake of completeness, we have also evaluated the overhead caused by the transformations performed by the DefinitionShadower. To do so, we have compared the

run time (JIT compilation and code execution) of the same test program, both enabling and disabling entity redefinition. This test program is comprised of a varying number of top-level declarations of different types (function, class or variable), ranging from 128 to 16384. Note that, in case of enabling entity redefinition, all the declarations have been given the same name. In order to obtain the overhead, we performed multiple executions and measured the average run time. All the executions were run on a platform comprised of 24 × Intel(R) Xeon(R) CPU E5-2695 v2 running at 2.40 GHz, and 128 GB of RAM.

As seen in Figures 6.a and 6.b, the run time behavior is similar for both, the function and class definition tests, yielding a time that grows with the number of declarations. Comparing the original Cling implementation (No shadow) with our proposal (Shadow), we can conclude that the "Shadow" version icurrs in an non-linear overhead in the range of 4—52%. One of the possible explanations for this variability, is that different LLVM/Clang data structure optimizations are applied depending on the entry size.

However, the test performed for variables (see Figure 6.c), while still growing with the number of declarations, it exhibits a much higher run time, with a smaller overhead ranging 2—13% for the "Shadow" version. This is due to the fact that wrapper functions generated around variable declarations call a Cling function that updates the internal state of the interpreter. Cling includes the `-noruntime` command line option that, among other things disables this behavior. As shown in Figure 6.c, the run time using this option is comparable to the other two cases. However, the overhead is much smaller, in the range of −28—15%, with the "Shadow" version being faster in some cases. Again, this variability might be caused by different optimizations in LLVM/Clang data structures.

At the light of the results, we can conclude that using the proposed technique allows interpreted C++ to obtain a closer behaviour to an interpreted language with moderate overheads.

7 Conclusion and Future Work

Interpreted languages have been proved to be a good solution for fast prototyping in agile development methodologies, and closing the gap between domain experts and application development. Since interpreted languages incur in extra overhead at runtime, in some cases it is necessary to generate a compiled version of the application. To pave the way, multiple implementations of C and C++ interpreters have been developed. Nonetheless, these languages present some inherent limitations due to their compiled nature.

In this paper, we present a technique to support entity redefinition in C++ interpreters, thus allowing the user to redefine functions, types and variables in a similar way to other interpreted languages such as Python. Relaxing the ODR aids rapid prototyping while keeping the C++ language relatively sound and consistent for the particular use case. Specifically, the contribution enables the following: *(i)* providing a new definition for a function; if the function is overloaded or templated, the new definition only overrides the matching overload candidate, thus not interfering with the C++ overload mechanism, *(ii)* overriding the definition of a type (`struct`, `class`, or `typedef`), while declared variables of an overriden type preserve the old type definition; therefore, old members of an overriden struct/class may be still accessed as long as a variable type is not refreshed redeclaring the variable, and *(iii)* providing a new definition for a global variable, possibly changing its original type or initialization value. Also, implementing this technique does not require major changes or patches to the compiler.

To validate the proposed technique, we have implemented the DefinitionShadower AST transformer to support entity redefinition in Cling with moderate overheads. As observed through the validation, entities can be given a new definition similarly to other interpreted languages. It is important to remark that the presented Cling implementation is part of the ROOT master branch, and will be used by the domain experts at CERN by the end of 2019.

As future work, we plan to address limitations of the current implementation, including allocated storage issues, and RTTI. Also, we intend to add more features to Cling, such as the generation of debugging information for JIT'ed code.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Economy and Competitiveness through Project Grant TIN2016-79637-P (BigHPC—Towards Unification of HPC and Big Data Paradigms).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[2] J. Akeret, L. Gamper, A. Amara, and A. Refregier. 2015. HOPE: A Python just-in-time compiler for astrophysical computations. *Astronomy and Computing* 10 (2015), 1 – 8. https://doi.org/10.1016/j.ascom.2014.12.001

[3] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. 2009. ROOT A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications* 180, 12 (2009), 2499 – 2512. https://doi.org/10.1016/j.cpc.2009.08.005 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.

[4] Erik Azar and Mario Eguiluz Alebicto. 2016. *Swift Data Structure and Algorithms*. Packt Publishing.

[5] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science and Engg.* 13, 2 (March 2011), 31–39. https://doi.org/10.1109/MCSE.2010.118

[6] Harry H. Cheng. 1993. Scientific Computing in the CH Programming Language. *Scientific Programming* 2, 3 (1993), 49–75. https://doi.org/10.1155/1993/261875

[7] QuantStack community. 2019. *Xeus Cling*. https://github.com/jupyter-xeus/xeus-cling/

[8] Steve Donovan. 2002. *C++ by example* (underc learning ed. ed.). Que, Indianapolis, IN. Accompanied by CD : CDR 01063.

[9] Michael Foord and Christian Muirhead. 2009. *IronPython in Action*. Manning Publications Co., Greenwich, CT, USA.

[10] Masaharu Goto. 1995. *C++ Interpreter - CINT*. CQ publishing.

[11] ISO. 2017. *ISO/IEC 14882:2017 Programming languages — C++*. ISO, 1214 Vernier, Geneva, Switzerland. 1605 pages. https://isocpp.org/std/the-standard

[12] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. 2010. *The Definitive Guide to Jython: Python for the Java Platform* (1st ed.). Apress, Berkely, CA, USA.

[13] Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (01 Jun 1968), 127–145. https://doi.org/10.1007/BF01692511

[14] Wim T.L.P. Lavrijsen and Aditi Dutta. 2016. High-performance Python-C++ bindings with PyPy and Cling. In *PyHPC 16: Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*. IEEE Press.

[15] Harri Luoma, Essi Lahtinen, and Hannu-Matti Järvinen. 2007. CLIP, a Command Line Interpreter for a Subset of C++. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 199–202. http://dl.acm.org/citation.cfm?id=2449323.2449351

[16] James Martin. 1991. *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA.

[17] Remigius Meier and Thomas R. Gross. 2019. Reflections on the Compatibility, Performance, and Scalability of Parallel Python. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2019)*. ACM, New York, NY, USA, 91–103. https://doi.org/10.1145/3359619.3359747

[18] Martin Odersky and al. 2004. *An Overview of the Scala Programming Language*. Technical Report IC/2004/64. EPFL Lausanne, Switzerland.

[19] T. E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science Engineering* 9, 3 (May 2007), 10–20. https://doi.org/10.1109/MCSE.2007.58

[20] Guido Rossum. 1995. *Python Reference Manual*. Technical Report. Amsterdam, The Netherlands, The Netherlands.

[21] PyPy Team. 2005. *Complete python implementation running on top of cpython*. Technical Report.

[22] V Vasilev, Ph Canal, A Naumann, and P Russo. 2012. Cling – The New Interactive Interpreter for ROOT 6. *Journal of Physics: Conference Series* 396, 5 (dec 2012), 052071. https://doi.org/10.1088/1742-6596/396/5/052071