Communication Papers of the 18th Conference on Computer Science and Intelligence Systems pp. 103–109
DOI: 10.15439/2023F4830
ISSN 2300-5963 ACSIS, Vol. 37

Compilation through interpretation

Adam Grabski
0000-0002-6283-8461
Warsaw Univeristy of Technology
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
Email: adam.gr@outlook.com

Ilona Bluemke
0000-0002-2894-5976
Warsaw Univeristy of Technology
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
Email: Ilona.Bluemke@pw.edu.pl

***Abstract*—As static metaprogramming is becoming more relevant, compilers must adapt to accommodate them. This requires exposing more information about the code, from the compiler to the programmer as well as more powerful compile-time function execution capabilities. The Interpreter component of a compiler therefore becomes more important.**

In this paper a novel approach to compiler architecture that places the Interpreter as the central component of the compiler is proposed. Translation of user code into the executable form is done by an Interpreter, written in the target language. The data structures of Interpreter are accessible also to the programmer. Such solution significantly improves flexibility and extensibility of the compiler and enables execution of any code at the compile-time.

Compile Time Function Execution (CTFE) First pattern was designed for low-level, non-garbage-collected, reflection-enabled language C-=-1. This is a new programming language, which allows the programmer to execute any code at the compile time, as well as to analyze and modify the program structure. Grace to the extensibility of the designed compiler, programs written in C-=-1 can also generate marshalling bindings for other languages and support a variety of programming paradigms.

The significant flexibility and extensibility of CTFEF caused severe problems in compiler construction. The compiler appeared very complex, its parts, especially Interpreter, were very difficult to debug. Compiler operates on inflexible data structures, accessible also for a programmer. They are therefore a part of the compiled languages standard library and backwards compatibility must be maintained.

I. Introduction

COMPILE-time function execution (CTFE) is an aspect of a programming language, that allows the programmer to execute code at compile-time. It has been gaining popularity with programming languages without virtual machines such as Rust [1] or C++ [2]. The goals and capabilities of CTFE depend on the language and are discussed in details in section II.

In this paper a new approach to compiler construction that places Compile Time Function Execution capabilities as the first priority is proposed. This architecture is called CTFEF: Compile Time Function Execution First. CTFEF builds the compiler around the Interpreter component. The goal of this approach is to shorten the initial stage of compiler bootstrapping [3], [4], better support languages built around static metaprogramming and make the compiler more extensible. The role of the compiler is to construct the semantic model of the program being compiled and pass it as data to the interpreted Compiler-Interface module. Compiler-Interface then transforms the model into the assembly language to be compiled into executable. This approach was created during implementation of the compiler for C-=-1, a new language that prioritizes static metaprogramming. Using CTFEF approach causes many implementation difficulties. Data structures used within the compiler are tightly coupled with the compiled language. These problems and how they were solved in C-=-1 compiler are described in section V.

The design of that language is described in section III. Related work is briefly shown in section II. In section IV the structure of a CTFEF compiler and interactions among its components are presented.

II. Related work

Many currently used programming languages allow the programmer to execute some code at compile time, for example: C# [5], [6], Rust [1], [7] and C++ [8].

Rust language compiler is the most similar in capability, to what was demonstrated with C-=-1, as a feature of CTFEF. It allows the user to write code that performs some transformations of the program, and interact with the environment during the build process. The two relevant features are build scripts and macros. Rust macros, similar to the classic C-style preprocessor macros, generate additional code as text. The major difference between the systems is that Rust macros operate on tokens, rather than text, and use syntax similar to regular Rust code. This mechanism is powerful, allowing the user to rewrite code to avoid repetition, simplify certain tasks and create new syntax. They are however unable to reflect on the program or obtain some information available to the compiler.

Build scripts, on the other hand, are programs that execute before the compilation of the main package. They prepare the environment for building the program, for example, compile external dependencies. The structure of the program being compiled, is not available for build scripts. Build scripts accessing such data, would have to analyze the code by themselves, without any compiler assistance, and that makes automatic generation of bindings for other programming languages very difficult.

C# compiler, called Roslyn [9], has the closest set of capabilities to what CTFEF aims to achieve. Its architecture allows for user-defined analyzers and code generators using a program model generated by the compiler [9]. These compiler extensions can be used and distributed as regular code

Topical area: Software, System and Service Engineering

packages, for example using nuget package manager [10]. C# code analyzers have access to entire analyzed code-base and to semantic analysis functionality of the compiler. Using CTFEF for the purposes of code analysis would not offer any additional benefits.

C# code generators are more limited. They can only add new files, not modify the existing ones. All code generators operate on a read-only snapshot of the code base. This also means that if more than one generator operates during compilation, they are unaware of the files created by other generators[11]. This limitation was introduced for a number of reasons. If source generators could influence each other, the order in which they run would become important and could lead to unpredictable behavior. Compilation performance would also be affected, as parallelization of source generators would become more difficult or even impossible.

Although CTFEF was inspired by the C# compiler, there are significant differences between them. C# compiler extensions are separate, compiled, dynamic libraries loaded by the compiler and executed as a regular code. In CTFEF components that analyze and generate code are part of regular code base and are not compiled separately. The intermediate representation of these modules is interpreted at compile time, together with other parts of compiler, and operate on the same representation of user program. CTFEF is further described in section IV.

III. Design of C-=-1

C-=-1 is a new programming language, designed as low-level and non-garbage-collected and compiled to native machine code, similar to C, C++ or Rust. What differentiates C-=-1 are its two core principles: all code is executable at compile-time and support for metaprogramming. The primary purpose of this language was to investigate how these ideas influence software written in it [12].

C-=-1 is a simple language, built with minimum set of features needed to demonstrate the usefulness of the proposed metaprogramming features. They are discussed further in section III-B. The primary motivation for those mechanisms was to provide to a programmer the ability to create domain specific static analysis and code generation tools, without creating a separate program.

A. Type system

C-=-1 type system is similar to type system in C++. Program may contain user defined classes, with members which may have limited accessibility. Generic programming is achieved by templates, although they are much more limited than the ones present in C++. The user may also use pointers to objects, with arbitrary indirection (for example pointer to pointer to object). Additionally, the language contains the concept of an `interface`, similar to the one found in C# [13].

B. Attributes and metaprogramming

Metaprogramming in C-=-1 is based on attributes. Attributes work in a manner similar to the ones found in C#. They are types, which may contain fields and methods. They can also be used to annotate other elements of the program, such as types, functions or variables.

In C-=-1 these attributes may implement special member functions that react to the use of an annotated program element. For example, an attribute that can be attached to a function, may implement `onCall` special member function. It will be called at compile time, for each invocation of the annotated procedure. Within the special method the attribute will have access to the semantic model of the call site. It may then modify the semantic model or report warnings or errors.

Listing 1 contains an example of a C-=-1 attribute providing static analysis: `noDiscard`. It works in the same manner as the attribute of the same name present in C++17 [14]: the result of invoking the annotated function must be used. To declare an attribute in C-=-1, the `att` keyword is used. After that, attribute targets should be listed in angled brackets, as in line 0 of Listing 1. Valid targets for attributes include a number of language elements, such as types, functions, variables and fields. Attribute from Listing 1 declares two member functions: `attach` in line 4 and `onCall` in line 6. Listing 2 contains an example of using the `noDiscard` attribute. All uses of the `noDiscardFunction`, except for the one on line 3 are valid. Using the function in a statement as opposed to an expression causes a compile-time error.

The `attach` method is called after names of all program elements have been gathered, but before compiler starts to analyze function bodies. It is common among all attribute targets and accepts the descriptor of the attached program element (function, field, type, etc.). Attribute can change aspects of the program that affect function overload resolution, such as whether a function is invokable at run or compile time, only from within the `attach` method.

The `onCall` method is an example of a function reacting to use an annotated program element. These methods are specific to a given attribute target. Within this function, the attribute may analyze and modify the code, as well as raise errors or warnings.

Listing 1: noDiscard attribute in C-=-1

```
public att<function> NoDiscard                                  0
{                                                               1
        public fn attach(f: functionDescriptor)                 2
        {}                                                      3
        public fn onCall(call:                                  4
            functionCallExpression*)
        {                                                       5
                if(call._parentStatment != null<                6
                    IInstruction>())
                    raiseError(                                 7
                    &(call._pointerToSource),                   8
                    "Return value of a no-                      9
                        discard function is not
                        used",
                    123                                         10
                    );                                          11
}                                                               12
}                                                               13
```

Lines 6 to 11 of Listing 1 are an example of a C-=-1 attribute providing static analysis. The `onCall` method checks whether

the attached function is invoked in an expression or instruction context. Calling a function as a statement means that the result of that invocation is discarded by the caller. This may indicate an error, when the function has no other side effects. If that is the case, the attribute calls the `raiseError` function, which is provided as a compiler intrinsic, that generates a compilation error. The example presented in Listing 1, although very basic, demonstrates the ability to implement a form of static analysis that typically requires modifying the compiler or creating an external tool.

Listing 2: Example of using noDiscard attribute from Listing 1

```
0   [noDiscard()]
1   fn noDiscardFunction() -> usize;
2   fn main() -> usize {
3         noDiscardFunction();
4         // error 123: Return value of
5         // a no-discard function is not used
6         let x = noDiscardFunction();     // ok
7         let y = x + noDiscardFunction(); // ok
8         return noDiscardFunction();      // ok
9   }
```

IV. Design of the compiler

CTFEF approach was created during implementation of the first compiler for the C-=-1 language[12]. CTFEF compiler has four major components:

1) Frontend.
2) Interpreter.
3) Compiler Interface.
4) Backend.

Figure 1 contains a diagram with an overview of how these parts interact with each other, during the compilation process. Frontend, described in section IV-A, parses the code in the compiled language and constructs its intermediate representation, using Interpreter's data structures. The structure and form of the intermediate representation is not specified by the CTFEF approach, as long as it contains all semantically relevant information from the source program. It is used to analyze both user code and the Compiler Interface. After the intermediate representation is constructed, it is passed to the Interpreter, which is described in section IV-B. Compiler Interface intermediate representation is then executed, using the user program as data. This step converts the semantic model of the program into the Backends intermediate language. This process is further explained in section IV-C. Finally, the Backend generates the executable file.

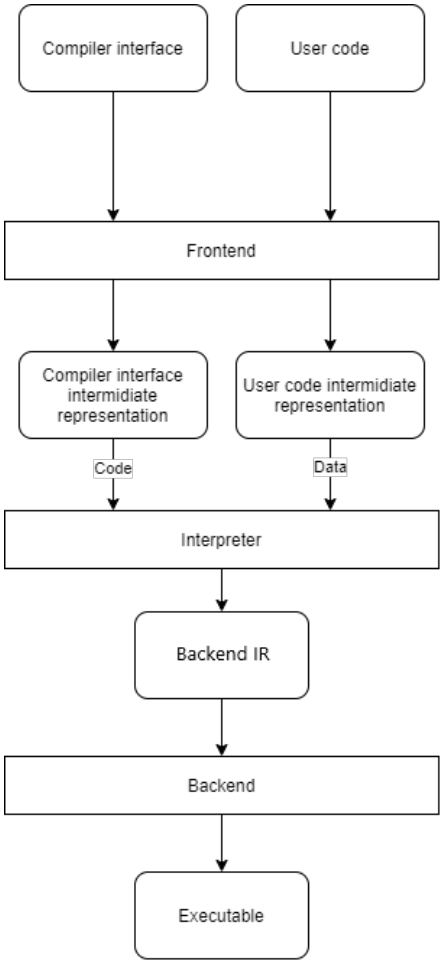

Fig. 1: CTFEF compiler structure

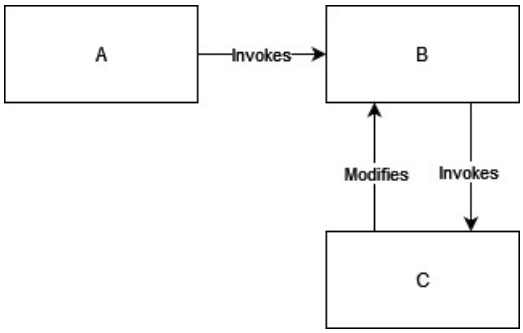

Fig. 2: Example of circular reference in meta code

A. Frontend

In the CTFEF approach, Frontend serves the same role of constructing the programs intermediate representation, as in conventional compilers [3]. The major difference lays in the data structures used to describe the program. For a CTFEF compiler, they must be accessible to the program running within the Interpreter. This may make Frontend more complex. The additional challenge of representing a user program, using Interpreter data structures, depends on the design of the Interpreter.

B. Interpreter

In CTFEF, Interpreter is main component of the compiler. It executes the Compiler-Interface which translates the intermediate representation into the Backend's assembly and serves as what is sometimes called the 'middle-end' of the compiler[15]. To do it, it must be able to treat the program's intermediate representation both as code and data.

An important issue for CTFEF compiler is decision what

code can be executed at compile-time. In some languages it may be possible to introduce circular references between the functions that modify the codebase. Figure 2 contains a diagram of such scenario. `A`, `B` and `C` are functions. If function `B` is modified by function `C` and `B` invokes `C`, the behavior of function `A` is unpredictable. This problem will only be magnified by larger program sizes.

One of possible solutions, to the above-mentioned problem, is to restrict which functions can be invoked at compile time. C-=-1 allows code within a compile time context to invoke procedures only from other packages, declared explicitly as dependencies. Circular references between packages, as in most other languages, are forbidden. C-=-1 additionally prohibits modification of dependencies. Therefore, it is impossible for a function to modify a procedure, it depends on.

C. Compiler Interface

Compiler Interface translates the program's intermediate representation into the Backend's assembly language. This component is interpreted during compilation and may be provided by the user. In case of C-=-1, compiler interface could be supplied to the compiler, the same way that the code being compiled is passed, as a collection of source files. Figure IV reflects this decision, treating the Compiler Interface as an input into the compiler, same as user code.

What is unique about CTFEF is that this part of the compiler can be written in the target language, during initial bootstrapping of the compiler bootstrapping process, i.e. initial implementation using another language [3], [4]. In case of the C-=-1 compiler, C++ was used to implement Frontend, Backend and Interpreter, with Compiler Interface written in C-=-1 [12].

Compiler Interface contains a function marked as the Compiler Interface Entry-point. That procedure must accept a set of modules to be compiled and a Compilation Context that is used to generate the Backends assembly. The module descriptors that are passed to the Compiler Interface are built by the Frontend, as can be seen in Figure 1.

After the Compiler Interface finishes generating Backend assembly, the Compiler Backend is invoked to generate the binary executable.

D. Backend

CTFEF does not put any additional requirements on compiler Backend. When using this approach, a generic Backend library can be used. C-=-1 compiler used LLVM[16] as its backend.

The Backend code must be invokable from within the interpreted program in the target language. Depending on how the Interpreter was designed, this may require significant effort. Compiler Backends are large and for the Compiler Interface to take advantage of them, their entire interface must be fully available in the interpreted context. This means exposing each function and type within the library to the interpreted code, by duplicating their signatures. These bindings could feasibly be generated automatically [17], but this technique was not used when implementing C-=-1 compiler.

V. Implementation

The first CTFEF was created for a new language: C-=-1, using generic parser generator and Backend. The most important aspect of implementing a CTFEF compiler is the design of the data structures, described in section V-A, that will be used by the Interpreter.

In order to exploit CTFEF approach in design of a compiler the language should contain a set of data structures to describe user code i.e. a Semantic Model. It will allow the programmer to interact and manipulate the structure of the program at compile-time. The Semantic model designed and implemented for C-=-1 is described in section V-B.

The final part of a CTFEF compiler is the Backend Interface. It is a program, written in the target language, and executed at compile-time that translates the semantic model into the Backends' assembly language. Backend Interface implemented for C-=-1 is relatively small and is described in section V-C.

A. Interpreter data structures

Data structures of the C-=-1 Interpreter have been designed having the ease of development and debugging in mind. They are thus not particularly efficient.

Figure 3 contains a class diagram of most of the types used to represent values within C-=-1. All of them derive from `IRuntimeValue` and are managed via C++ smart pointers. The interface of the base class allows the value to be converted to a human-readable format, serialization, deserialization and copying.

The most primitive types within the hierarchy are `StringValue` and `IntegerValue`. They are simple wrappers for strings and integers, present in host language. Floating point numbers were not implemented as they were not necessary for implementation of a basic compiler.

User-defined types are represented using `ObjectValue`. The contents of an object is kept as a `string - IRuntimeValue` dictionary, with field names as keys and `uniqe_ptr< IRuntimeValue>` as values. C-=-1 object is therefore spread out in memory, even if the fields are directly contained within the class, without any indirection.

There are several types of reference within the C-=-1 Interpreter. The most basic pointer type is a reference to C-=-1 value. It was realized as a pointer to the owning pointer of the value.

B. Program semantic model

A major motivation for creating CTFEF was the ability to support languages with compile-time metaprogramming. This includes reflection and modification of the code being compiled. User program has to be represented as a complete and modifiable object, using the Interpreters' data structures.

C-=-1 language model divides the user program into assemblies. They represent an individual program package: a library or an executable file. The compiler is invoked to compile an assembly, together with its dependencies. Assembly is the root object of C-=-1 program model, it stores a list of assemblies it depends on and the root namespace of the package it

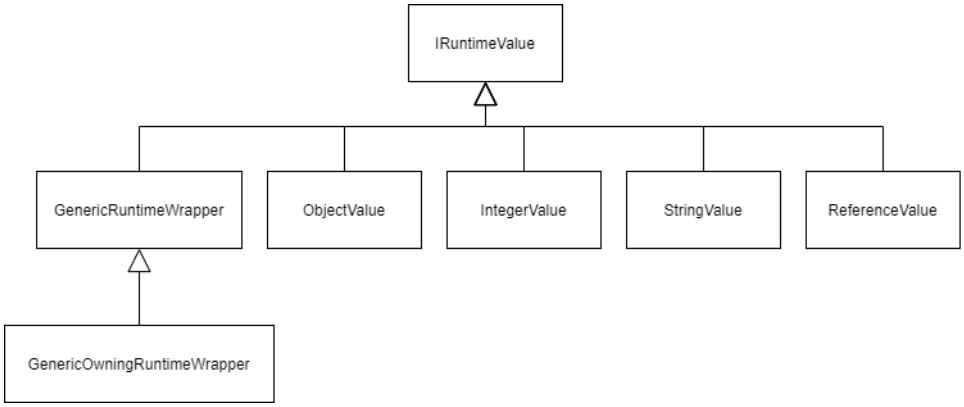

Fig. 3: Class diagram of C-=-1 Interpreter data structures

represents. The remainder of user code is organized into namespaces, types, functions and fields. These parts of the model are represented by native classes of the host language and form the basis for the rest of the model.

The most complex part of the model is the representation of the function body. Like in most programming languages, a C-=-1 function can contain a variety of instruction types. That includes complex statements and blocks of statements that can be arbitrarily nested. Each instruction may also contain expressions of any complexity.

To deal with this complexity, C-=-1 semantic model for functions is build around two interfaces: `IInstruction` and `IExpression` and their concrete implementations. Every category of instruction or expression is represented by its own type. The user may then analyze the structure of the program, using a dynamic type conversion mechanism similar to C++ `dynamic_cast` [2].

All elements of the semantic model, have a `sourcePointer`. It is a simple structure, that contains the filename and the line number of the expression or instruction. This information can be passed to compiler intrinsic functions, such as `raiseError`, to generate error messages for the user. Listing 1 contains an example of this functionality. The `pointerToSource` makes the messages generated by the compiler easier to understand for the programmer.

Component responsible for creating the semantic model is a major part of the compiler. There are two operations that this module performs: building the definition of the types used to describe a program, and creating an instance of the model, given semantic information. C-=-1 compiler has a hard-coded definition of its base library. It contains definitions of primitive types and types used to build the semantic model of a program. The description of this library must be built manually, as it is very closely integrated with the compiler.

C. Backend interface

The C-=-1 Backend interface uses LLVM [18] code generation API that has been exposed by the compiler. The functionality which has been made available to C-=-1 represents a minimal subset of LLVM, that is sufficient to implement a basic compiler.

Besides translating user code, the Backend Interface must also generate the assembly for certain intrinsic operations. Functions such as integer arithmetic operators, array indexers or memory allocators are concepts too low-level to be expressed in C-=-1. They are therefore expressed as functions, without bodies, which are then replaced by appropriate intrinsic operations.

Listing 3 contains a simple function written in C-=-1 (line 2), its ideal LLVMIR (line 5) and LLVMIR generated by C-=-1 compiler (line 21). The current implementation generates a function for each operator, regardless of whether it was defined by the programmer or is a primitive operation. They are merely wrappers around the actual LLVM intrinsic, meant to simplify implementation of the Backend Interface. Future versions, with additional effort, may generate the ideal LLVMIR from line 21 of Listing 3.

Certain other intrinsic operations are defined using external dependencies. C-=-1 memory management library, in the runtime context, uses a simple interface capable of allocating and deleting a continuous buffer. It consists of two functions: `unsafe_new` and `delete`. In the standard library, they are explicitly mapped to `malloc` and `free` functions from the C runtime.

Backend interface must also allow the programmer to influence how the executable code is generated. There are many practical reasons for this capability. Specifying the name of a function in order to link it to an external symbol is one of them. For example, C-=-1 standard library uses `malloc` and `free` to manage memory. These symbols are imported as `unsafe_new` and `delete` in excerpt in Listing 4. This is accomplished using the `mapToExternalSymbol` attribute and specifying the symbol name as a parameter, as was done in lines zero and three.

One of possible ways of achieving this, is to declare an interface for an attribute generating a functions symbol name. Listing 5 contains relevant code of a Backend Interface that uses such an attribute to override mark external symbols. Interface `ISymbolNameOverride` contains only one method:

`createSymbolName` that returns the name of the symbol in the generated assembly.

Listing 3: Representation of average function in C-=-1 and LLVM IR

```
1  // C-=-1 function in source code
2  fn average(a: usize, b: usize, c: usize) -> usize {
3    return (a + b + c) / 3;
4  }
5  // Ideal representation in LLVMIR
6  define i32 @average(i32 %0, i32 %1, i32 %2){
7    %4 = add i32 %1, %0
8    %5 = add i32 %4, %2
9    %6 = sdiv i32 %5, 3
10   ret i32 %6
11 }
12 // Generated LLVM IR
13 define i32 @__operator___plus____usize__usize(i32
       %0, i32 %1){
14   %3 = add i32 %1, %2
15   ret i32 %3
16 }
17 define i32 @__operator___div____usize__usize(i32 %0,
       i32 %1){
18   %3 = sdiv i32 %1, %2
19   ret i32 %3
20 }
21 define i32 @average(i32 %0, i32 %1, i32 %2){
22   %4 = call __operator___plus____usize__usize(i32
         %1, i32 %0)
23   %5 = call __operator___plus____usize__usize(i32
         %4, i32 %2)
24   %6 = call __operator___div____usize__usize (i32
         %5, i32 3)
25   ret i32 %6
26 }
```

Functions `buildFunction` and `getFunctionName`, from Listing 5, are parts of Backend Interface. They are invoked in order to convert a C-=-1 `functionDescriptor` to a LLVMIR function. They were included in the Listing, because they are the only parts of the Backend Interface that need to interact with `ISymbolNameOverride` attributes. Both of these procedures, check whether an attribute implementing this interface is attached to the function they are currently processing. This happens on lines two and eighteen of Listing 5. If that attribute is present, code of that function is ignored, and it is treated as an external symbol: condition on line eighteen omits execution of `build_block` function on line twenty-one. Procedure `getFunctionName` contains a similar condition on line two, that decides how the name should be generated. If an `ISymbolNameOverride` attribute is present, it will be created by `createSymbolName` method of the attribute attached to the function. Otherwise, the `mangleName` function will create name of function's symbol based on its parameters and return type.

Listing 4: C-=-1 memory allocation functions from standard library

```
0  [mapToExternalSymbol("malloc", "")]
1  private fn unsafe_new(size: usize) -> char* {}
2
3  [mapToExternalSymbol("free", "")]
4  internal fn delete<typename T>(val: T*) {}
```

Listing 5: Part of a Backend Interface, using `ISymbolNameOverride` interface

```
0  private fn getFunctionName(f:
       functionDescriptor) -> string {
1      let attribute = f.get_attribute<
           ISymbolNameOverride>();
2      if(attribute != null<
           ISymbolNameOverride>())
3        return attribute.createSymbolName();
4      return mangleName(f);
5  }
6  private fn buildFunction(
7      f: functionDescriptor,
8      llvmF: llvmFunction,
9      registry: packageRegistry*,
10     mod: llvmModule)
11 {
12     let variables = dictionary<
           variableDescriptor, llvmValue>();
13     let params = f.parameters();
14     for(i in enumerate(0, params.length()))
15       variables.push(params[i], llvmF.
           getParameter(i));
16     let builder = llvmF.getBuilder();
17     let attribute = f.get_attribute<
           ISymbolNameOverride>();
18     if(attribute == null<
           ISymbolNameOverride>())
19     {
20       let code = f.code();
21       build_block(&code, &builder, &
           variables, registry);
22     }
23 }
```

VI. Conclusions

CTFEF is a new approach to compiler construction which offers high degree of compiler extensibility, at the cost of development time and performance, compared with conventional compilers. It places the interpreter as the main component of the compiler and focuses on executing user code at compile time. The user code has access to the same information as the compiler at compile time and may perform analysis or transformation of the compiled program. The thesis that introduced this approach [19] demonstrated numerous practical application: generating bindings for other languages, static analysis and extending semantics of the language. These goals are significantly easier to accomplish, thanks to the access to semantic model of the program, constructed by the compiler. Authors of language tools do not need to analyze the user program.

On the other hand, using CTFEF approach has some drawbacks. Implementing a compiler is more difficult. Since a significant part of the compiler is interpreted, the initial implementation requires working with the target language. The lack of available tools, such as integrated development environments, debuggers and libraries, for this new language, makes this part of the process significantly more difficult. On the other hand also means the work on the compiler in the target language may start at the very beginning of compiler bootstrapping process[3], [4].

The compiler created for C-=-1 has unacceptable performance, as noted by the original C-=-1 paper [12]. Compiling the C-=-1 standard library, containing around 200 lines of code, takes approximately 10 minutes. Majority of that time is spent on interpreting the compiler interface. This is a

significant barrier to adopting CTFEF approach. Since the compiler implemented for C-=-1 was made as a research tool with minimal effort, further work is needed to explore the performance issues of CTFEF approach.

REFERENCES

[1] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[2] "Programming languages — C++," International Organization for Standardization, Geneva, CH, Standard, Mar. 1998. [Online]. Available: https://www.iso.org/standard/25845.html

[3] A. Puntambekar, *COMPILER DESIGN*. Technical Publications, 2011.

[4] D. Novillo, "Gcc internals," in *International Symposium on Code Generation and Optimization (CGO), San Jose, California*, 2007.

[5] Source generators. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview

[6] Dotnet. Roslyn. [Online]. Available: github.com/dotnet/roslyn

[7] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[8] "Programming languages — C++," International Organization for Standardization, Geneva, CH, Standard, Mar. 2020. [Online]. Available: https://www.iso.org/standard/79358.html

[9] N. Vermeir, ".net compiler platform," in *Introducing .NET 6*. Springer, 2022, pp. 275–295.

[10] M. Balliauw and X. Decoster, "Nuget package manager console powershell reference," in *Pro NuGet*. Springer, 2013, pp. 331–338.

[11] B. O. SLIMÁK and R. J. Pelikán, "Source generators in c#," Master's thesis, Department of Computer Systems and Communications, Masaryk University, 2022.

[12] A. Grabski, "Compilation through interpretation: static metaprogramming in c-=-1," Master's thesis, Warsaw University of Technology, 2022.

[13] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[14] "Programming languages — C++," International Organization for Standardization, Geneva, CH, Standard, Mar. 2017. [Online]. Available: https://www.iso.org/standard/68564.html

[15] M. Hsu, *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries: Design powerful and reliable compilers using the latest libraries and tools from LLVM*. Packt Publishing, 2021.

[16] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[17] P. Dietz, T. Weigert, and F. Weil, "Formal techniques for automatically generating marshalling code from high-level specifications," in *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998, pp. 40–47.

[18] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 427–440. [Online]. Available: https://doi.org/10.1145/2103656.2103709

[19] A. Grabski, "Rose parser generator," Bachelor's thesis, Wydział Elektroniki i Technik Informacyjnych, Politechnika Warszawska, 2020.