Code-driven Language Development: Framework for Analysis of C/C++ Open-Source Projects

Bachelor Thesis

Author(s):
Hartogs, Siegfried

Publication date:
2021-03-28

Permanent link:
https://doi.org/10.3929/ethz-b-000480835

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Code-driven Language Development: Framework for Analysis of C/C++ Open-Source Projects

Bachelor Thesis

Siegfried Hartogs

March 28, 2021

Advisors: Prof. Dr. T. Hoefler, M. Copik

Department of Computer Science, ETH Zürich

Contents

Contents **i**

1 Introduction **3**
1.1 Goal and motivation . . . 3
1.2 Research questions . . . 3
1.3 Organization of the thesis . . . 4
1.4 Example . . . 4

2 Features **5**
2.1 C++ language features . . . 5
2.1.1 Core language features . . . 6
2.1.2 Templates . . . 12
2.2 C++ Standard Library . . . 14
2.3 Deprecated and unsafe constructs, best practices and anti-patterns . . . 15
2.4 General-purpose features . . . 16
2.5 Software metrics . . . 16

3 Framework **17**
3.1 Tool . . . 17
3.2 Framework . . . 18
3.2.1 Stages of the framework . . . 18
3.3 Analysis internals . . . 20
3.3.1 Emitting features from code . . . 20
3.3.2 Emitting statistics from features . . . 22
3.4 Interfacing with Clang . . . 23

4 Analyses **25**
4.1 Implemented analyses . . . 25
4.1.1 Cyclomatic Complexity Analysis (CCA) . . . 25
4.1.2 `constexpr` Analysis (CEA) . . . 25
4.1.3 Loop Depth Analysis (LDA) . . . 26
4.1.4 Loop Kind Analysis (LKA) . . . 27
4.1.5 Function Parameter Analysis (FPA) . . . 28
4.1.6 Template Instantiation Analysis (TIA) . . . 29
4.1.7 Move Semantics Analysis (MSA) . . . 35
4.1.8 Template Parameter Analysis (TPA) . . . 37
4.1.9 `using` Analysis (UA) . . . 38
4.1.10 Variable Template Analysis (VTA) . . . 39
4.2 Testing methodology . . . 40
4.3 Known issues . . . 41

5 Results of the Analysis of the LLVM Core Libraries **43**
5.1 Results 43
5.2 Discussion 46

6 Conclusion **51**

A Appendix **53**
A.1 Full list of containers and algorithms analyzed 53
A.2 Flags for `cxx-langstat` 53

References **55**

Abstract

C++ has substantially grown during the last ten years and features such as move semantics, parameter packs, and keywords such as `constexpr` have been added. Along with that come guidelines to write correct and maintainable C++ code. While there has been work describing the adoption of features in C++ code, they seem to typically employ a specialized tool to analyze source code, making them inadequate to analyze new language features in the future.
This thesis overcomes that limitation by building a framework that can be used to write analyses on top of, be it to study new keywords or the adoption of programming guidelines. We achieve this by leveraging Clang to provide us with the abstract syntax tree (AST) of the input code. Our tool has the advantage that it is fit to analyze upcoming features since future versions of Clang will parse the source code for us. This simplifies research about the adoption of features by avoiding the technicalities of parsing source code and should be accurate thanks to the rich representation of C/C++ in the AST. We demonstrate the results of the tool by showing insights gained about – among other features – the adoption of range-based loops, parameter packs, and C++ Standard Library containers and algorithms.

Keywords: C++, language features, library usage, static code analysis, abstract syntax trees, Clang

Acknowledgements

I want to thank Marcin Copik and Torsten Hoefler for their assistance and valuable feedback in writing this thesis.

Chapter 1

Introduction

1.1 Goal and motivation

C++ is a very rich language with many paradigms and features, and there have been many papers discussing how features of it are adopted and used, see e.g. [18] on how C++ templates are used for generic programming and [16] on the adoption of lambdas. This thesis describes a new approach that tries to simplify such research about features in C/C++ by proposing a general framework that supports adding new analyses easily, allowing to focus on the analysis of the language features instead of the infrastructure. We achieve this by building the framework around Clang, the frontend of the LLVM compiler. We can leverage Clang by using it as a library to lex and parse source code for us, giving us directly the abstract syntax tree of a piece for source for further analysis. Furthermore, Clang also provides us with so-called AST matchers, which greatly aid in finding relevant locations in the AST and thus in source code.

1.2 Research questions

The aim of the thesis is to build a framework that serves as a basis for implementing analyses that detect and measure language and code features in C/C++. These analyses may address questions such as:

RQ1. Which new language features are easily and quickly adopted: range-based loops, move semantics, new keywords? Are there new standard components that are rarely seen in codes?

RQ2. What is the usage rate of new standard modules and libraries, such as smart pointers and new containers?

RQ3. C++ comes with a large library of standard algorithms, but they are rarely taught and the common perception is that they're not commonly used. Which algorithms are the most popular, which are not used at all? The usage of algorithmic skeletons can help with automatic parallelization and porting of code to GPUs.

RQ4. Which non-standard libraries are most commonly spotted in projects? Can we deduce recommendations for new directions of standard development, based on the most commonly needed features, such as JSON support or serialization?

RQ5. How many projects involve any form of parallel processing? Which parallel frameworks are commonly spotted? `std::thread`, OpenMP, TBB, others?

RQ6. There are many unsafe and deprecated language features, as well as bad programming patterns. Many of them can be detected by code linters and static analyzers, but these tools might not be commonly used. In how many projects can we detect the presence of such features? How often do projects adapt `override`, `final` or `noexcept` keywords; how many functions should use them, but don't? Might it be

a common problem, recommending further actions to improve the quality of C++ codebases?

RQ7. Which types of new features have the highest impact on new codebases and which are frequently adopted by projects that have been in development for a long time?

The main question this thesis investigates is whether it is possible to build a general framework that we can use to implement analyses that analyze different features of C/C++ code. I will try to answer this question by proposing a framework that can address the above questions.

1.3 Organization of the thesis

For an easy entry, we start with a small motivational example describing the process of using this framework to gain knowledge about the prevalence of features. Chapter 2 makes a not-so-brief introduction into a variety of features and our motivation for analyzing them. Chapter 3 details the inner workings of the framework and how it interacts with the analyses that it supports. Chapter 4 lists a selection of analyses implemented on top of the framework. Chapter 5 details results gathered on a real-world project, namely the core libraries of the LLVM project, and discusses them to address the research questions from the previous section.

1.4 Example

Conceptually, these are the steps needed to create an analysis on top of this framework:

1. Identify the language property of interest, for example, structured bindings. The aim is to learn about its adoption by analyzing what percentage of variables declared come from structured bindings.
2. Extraction: in order to answer that question, extract data describing two features:
   - variables declared by structured bindings
   - variables not declared by structured bindings

   It is possible to extract more information here, e.g. the types of the variables in the structured bindings, which can help to understand why structured bindings were used.
3. Counting: now that raw data is available, it is possible to implement a function that counts the two kinds of variables to get the relative popularity of structured bindings.
4. Register the analysis with the framework.
5. Choose one or more C++ projects and run the tool on them. For example, one can analyze multiple versions of the codebases to see if structured bindings get more popular the longer they are a part of C++.

Chapter 2

Features

When we say that a language gains a new feature, we usually mean that the language designers have added a new language construct or library function that makes programming easier or more elegant. Looking at a piece of code that employs that new language feature, one can usually discern the locations where it is used.

We generalize a feature to be a predicate that describes a set of code locations with some property. Such a predicate can describe a myriad of patterns in code.

Definition 2.1 (Feature) *A feature is a predicate that describes*

- *a language construct, e.g. function declaration or loop statement,*
- *a language keyword,*
- *an idiom,*
- *a best practice,*
- *an anti-pattern, i.e., a bad programming practice,*
- *a particular usage of a programming library.*

To get statistics on language usage we extract features from code, meaning that we get all locations that use that feature, i.e., satisfy its predicate. Each single code location in that extracted data is an instance or occurrence of that feature.

In this background chapter, I want to introduce a variety of features that are interesting and have the potential to be analyzed. I briefly describe a selection of features ranging from C++ language features, programming idioms, software metrics to C++ libraries and best practices.
For all details about the C++ language and its Standard Library (sections 2.1, 2.2), see [4] and www.cppreference.com.

How these features are extracted from real-world code and what insights we gain from them will be discussed in subsequent chapters.

2.1 C++ language features

We start with what first comes to mind when one talks about features in the context of programming languages, namely the core language constructs of C++.

C++ encompasses multiple paradigms like procedural, object-oriented, and even functional programming for example through template metaprogramming.

We first describe some core C++ language features that were added since C++11 and continue by discussing C++ templates (arguably, these are also part of the core of C++, but important enough to deserve their own section).

2.1.1 Core language features

References, value categories and move semantics

A reference declares an alias to an object or function. References, unlike pointers, cannot be "null", they always have to refer to something. It is, however, possible to write code s.t. the reference becomes invalid, e.g., by having a reference to a local variable that goes out of scope. A reference, once declared, cannot be reseated, i.e. it has to refer to the same thing throughout its lifetime. References aren't objects, there thus are no references to references [24]. When a function takes in a parameter by reference, it means that the parameter refers to the argument given at the call site. This is where the trivial pattern-matching I always used to perform in my head breaks: the type of the argument does not have to exactly match the type of the function parameter:

```
void f(int& i){
    i++;
}
int x = 0;
f(x);
// x has type int, but i has type reference-to-int
```

References are, depending on whether they refer to an lvalue or rvalue, indicated at the type with a single or double ampersand, respectively, e.g., `int&` (lvalue reference to `int`) or `float&&` (rvalue reference to `float`). Forwarding references (sometimes also called universal references) are also written with a double ampersand, but we'll defer those to a later section.

Lvalues and rvalues An expression can be split up into two *value categories*. Omitting details to get the intuition across, those are the two following: An lvalue is a value that can be referred to by name and that can be assigned to, such as an expression on the left-hand side of the assignment operator "=". An expression on the right-hand side of "=" can be either lvalue or rvalue. Rvalues often indicate temporaries or literals, and cannot be referred to by name. An intuition to dinstinguish the two is that one can take the address of lvalues, but not of rvalues.
The type and the value category of an expression are two orthogonal concepts, exemplified using an example similar to [41] (p. 2, 3).

```
void g(int&& i);
// i is an lvalue (it has a name, you can refer to it),
// but its type is rvalue reference (i.e. reference to rvalue)
```

For a more in-depth explanation, see[1] and see[2] for a more forgiving explanation about value categories and move semantics.

Lvalue references When calling a function in C++, the arguments are passed by copy, i.e., in the callee's scope a new variable will be created, thus any operations will leave the value of the caller unchanged:

```
void f(int i){
    i++;
}
int x = 0;
f(x);
// x is still 0
```

[1] https://en.cppreference.com/w/cpp/language/value_category
[2] https://stackoverflow.com/questions/3106110/what-is-move-semantics

However, when using lvalue references, the callee receives a reference to the argument the caller called with, thus the caller will modify the original value:

```
void f2(int& i){
    i++;
}
int x = 0;
f2(x);
// x is now 1
```

Rvalue references Since rvalues are temporaries, they allow for an interesting optimization: *move semantics*. Calling a function that has an rvalue reference parameter means that the function is allowed to alter the object the reference refers to in any way it likes - without having to worry about anybody else needing the corresponding data later. This is especially handy in two situations:

Avoiding copies using move constructors As we know from the previous paragraph, passing by value causes a new object to be created. Prior to C++11, we were always required to use the copy constructor, incurring a penalty due to having to copy the object.

When the object manages memory through a pointer, that probably means performing a deep copy of that managed memory instead of only copying the pointer, making it expensive due to additional memory allocations.

With rvalue references we know that the reference refers to a temporary object, enabling us to initialize the parameters of a function using the move constructor. This means that the parameter can take over the argument's externally managed resources by setting the pointers of the parameter to point to the memory pointed to by the pointers of the argument.

Notice that resources that are not externally managed – like integral fields and the like – still have to be copied over. The advantage of the approach is that – without having to resort to reference function parameters – the object will be copied when necessary, but can be moved when not. To give a better idea, I will go through an example (see listing 2.1) step-by-step:

1. Notice how `C()` on line 26 calls the default constructor of `C` from line 4 – its result is unnamed, and thus an rvalue.
2. The call of `func` on line 26 causes the compiler to figure it has to construct `c` on line 22 from the argument `C()`.
3. Realizing `C()` is an rvalue, it invokes `C`'s move constructor from line 10 to construct `c` from `C()`.
4. Using the move constructor, `c`'s `b` is set to the value of `C()`'s `b` (line 11) and `C()`'s `b` is set to `nullptr` (line 12).
   The second step is necessary because when `C()` goes out of scope and its destructor gets called (line 14), the `b` that used to belong to it must not get destroyed with it since `c` now needs that data.

(Admittedly, I'm simplifying a bit here: until C++17 the call of move constructor is often optimized out by directly constructing `C()` into `c`'s storage. Since C++17 that so-called *copy elision* is mandated by the language [23, 21].)

```
class C {
public:
    // default constructor
    C() {}
    // copy constructor
    C(const C& c) {
        // Perform deep copy of Big, assumed to be expensive
    }
    // move constructor
    C(C&& c) {
        b = c.b;
        c.b = nullptr;
    }
    ~C() {
        delete b;
    }
    // move doesn't help
    int x;
    // move helps
    Big* b = new Big();
};
void func(C c){
    // do something with c
}
//
func(C());
```

Listing 2.1: Class with move support

To explicitly instruct the compiler to move from an lvalue, one can use `std::move`:

```
C c;
func(std::move(c));
```

Since `c` is an lvalue, the copy constructor would be invoked to create the function parameter. Using `std::move` will cast it to an rvalue reference, enabling to use the move constructor.

Move-only types Thanks to C++11's introduction of a sharp boundary between copying and moving, a possibility for a new kind of type appears: move-only types. Move-only types, as the name suggests, cannot be copied, only moved. The prototypical example of that is `std::unique_ptr`. Its semantics dictate that it should be the only smart pointer to point to and manage an object, making it the owner. When a `std::unique_ptr` object it destroyed, the object it manages will be deallocated with it, liberating the user from having to call `delete`, as well as ensuring that the memory will be reclaimed in case of an exception. Given that double-frees lead to undefined behavior, it is important that only one `std::unique_ptr` to a region of memory exists, thus making the smart pointer uncopyable.

`const` lvalue references Usually, lvalue references will only bind to lvalues, rvalue references will only bind to rvalues. The exception are `const` lvalue references, which will bind to both lvalues and rvalues. Imagine C++ `const` lvalue references couldn't bind to rvalues. That would require one to write two overloads for the copy constructor, even though they should not modify the object that is to be copied:

```
class C {
public:
    // copy constructors
    C(const C& c) { ... }
    C(C& c) { ... }
    ...
};
```

`const` rvalue references are rarely useful: being unable to *take over* their resources, you wouldn't be able to properly move from them.

In conclusion, the C++ language has spawned four useful ways of passing data:

- by value/"copy"
- by non-`const` lvalue reference
- by `const` lvalue reference
- by rvalue reference

Forwarding references

Forwarding references (sometimes called universal references) were, like rvalue references, added to C++11. Syntactically, they look the same as rvalue references, and can occur in function template parameter lists or as variable declarations with `auto` type deduction [24].

```
template<typename T>
void f(T&& t);

auto&& w = Widget();
```

Their goal is to enable *perfect forwarding*, which aims to "forward" a function argument by preserving their value category using the `std::forward` function template [24]. Example from [25]:

```
1 template<typename T>
2 void wrapper(T&& arg){
3     foo(std::forward<T>(arg));
4 }
```

If `arg` (line 2) refers to an lvalue, then `foo` should be called with an lvalue, not with an rvalue, which can be dangerous (e.g., `foo` might try to move from `arg`, which is problematic if what `arg`'s bound to is still needed later). If `arg` refers to an rvalue, then we want to call `foo` with an rvalue s.t. `foo` may move from it.
For example, when `wrapper` is called with the literal `10`, then `t` binds to an rvalue and `T` is deduced to `int`. If we first define a variable `int a = 10` and call `wrapper(a)`, then `t` binds to an lvalue and `T` is deduced to `int&`.
Whereas `std::move` unconditionally casts to rvalue reference, `std::forward` is used to preserve the value category of the argument.

To distinguish rvalue and forwarding references at the parameter list of a function template, one can check the following two conditions of `T&&` as stated in [40], [41] (p. 168).

- `T&&` must be *exactly* `type&&`, not `std::vector<type>&&` or similar; no qualifiers such as `const type&&`
- `T` must be deduced when the function template is *called*

Function templates include member function templates: One has to be careful not to misread the references here (listing 2.2), `t` (line 3) is not a forwarding reference, but an rvalue reference. The reason for that is that `f1` is part of a class template. When `f1` is called, it has to be part of some class specialization, meaning that `T` has been deduced

```
1 template<typename T>
2 class C {
3     f1(T&& t){
4     }
5     template<typename U>
6     f2(U&& u){
7     }
8 };
```

Listing 2.2: && references in member functions

during the instantiation of the class, not during the call.
By contrast, `u` on line 6) is a forwarding reference because `U` is deduced during the call to `f2`.

Variable declarations that employ forwarding references can mostly be easily discerned thanks to `auto&&` [24].

(We're simplifying here, to see how in reality forwarding references are rvalue references where *reference collapsing* occurs, see [41] (p. 197).)

Type definitions and type aliases

C++ inherited type definitions (`typedef`) from C, which is one of two ways to create names that abbreviate complex types. This is useful in C++ because types can become very long and frustrating. Because C++ never added any means to parameterize type definitions, the so-called "`typedef` template" idiom [41] (p. 63) emerged, a workaround to template type definitions. It exploits the fact that when the definition of a class member is needed, the compiler will create an instantiation of it, which in our case is a type definition:

```
// typedef
typedef std::vector<int> MyIntVector;

// "typedef template" idiom
template<typename T>
struct MyVector {
    typedef std::vector<T> type; // doesn't have to be named "type"
};
// Use it like so:
MyVector<int>::type v = {1,2,3};
```

We had to wait until C++11 to get a neat solution: type aliases. Not only did they make declaring synonyms more intuitive by introducing new syntax, but also allowed them to be templated, finally making the old idiom obsolete. Non-templated type aliases are semantically equivalent to type definitions [35].

```
// alias
using MyIntVector = std::vector<int>;

// alias template
template<typename T>
using MyVector = std::vector<T>;
// Use it like so:
MyVector<int> v = {1,2,3};
```

It is interesting to note that Herb Sutter proposed to add real `typedef` templates to C++ [42], which unfortunately were rejected.

Range-based `for` loops

C++11 added next to the already existing `for`, `while` and `do-while` loops the so-called range-based `for` loop (also called `for`-each loop or range-`for`). It allows the programmer to intuitively iterate a range from beginning to end without the hassle of having explicit indices or pointers, e.g. when iterating over a container:

```
std::vector<int> v = {1,2,3};
for(auto i : v){
    // do something with i
}
```

According to [19], it is syntactic sugar for using iterators and dereferencing the pointer. Not all old-style loops can be converted to range-based loops though, such as:

- infinite loops
- loops that invalidate iterators midway, e.g. loops that modify containers through `push_back` or `erase`

This should be kept in mind when assessing the adoption of range-based `for` loops.

`constexpr`

`constexpr` is a C++11 keyword that variables, (member) functions, constructors and, more recently since C++20, even destructors can be ameliorated with. It has an interesting, albeit a tad complicated meaning:

- for variables a `constexpr` specifier means that it is possible to compute its value at compile-time. `constexpr` variables are implicitly `const` [30].
- when a `constexpr` function is called with only compile-time arguments, the result of the function will be computed at compile-time. If, however, any argument is not known at compile-time, the computation will be executed at runtime, like a regular function.

This blurring of the compile-time-runtime boundary allows calculations that were traditionally executed during runtime to be performed when compiling, which may lead to faster running code [41] (p. 101).

C++14 pushed the envelope by allowing `constexpr` functions to be more intuitive, e.g., by allowing them to contain loops and multiple return statements [41] (p. 100).

C++17 added `if constexpr`, which requires that the condition is a value known at compile-time. This can be leveraged to discard either branch of the `if` statement. An example taken from [22]:

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t;  // deduces return type to int for T = int
}
```

Knowing the relevant branch at compile-time means that not all branches need to be compiled, leading to shorter compilation times and smaller executables.

2.1.2 Templates

In this section I will describe the basics of C++ templates (which were added in C++98), but also touch on more recent additions to templates like parameter packs and variable templates, which were only introduced in C++11 and C++14, respectively.

Templates describe a family [27]:

- class templates describe a family of types,
- function templates describe a family of functions,
- variable templates describe a family of variables,
- alias templates describe names that refer to a family of types [28].

Templates reduce programming burden and increase maintainability by allowing programmers to write a piece of logic only once and then to "fill it in".

Template parameters

A template is parameterized by template parameters, which can be one of three [33]:

- non-type template parameter,
- type template parameter,
- template template parameter.

One of each is show below:

```
// Template
template<int N, typename T, template<int> typename C>
class V {
    C<N> c;
};
```

Template parameters don't have to be named, see the `int` in template template parameter `C` for example. This is useful when the parameter is not relevant, as for example with forward declarations.

Parameter packs A parameter pack is a special kind of parameter that accepts zero or more arguments. A pack can be a pack of non-types, types, or templates.

```
template<typename... Ts> // Ts is a template parameter pack of types
void f();
```

There are two kinds of parameter packs, namely template parameter packs and function parameter packs [26]. Template parameter packs occur in a template parameter list, i.e. between the "`< >`" of a template declaration. By contrast, a function parameter pack occurs in the parameter list of a function signature, which is neat to write functions that accepts multiple arguments – like so:

```
template<typename... Ts> // Ts is template parameter pack
void f1(Ts... ts); // ts is function parameter pack
```

A template is called variadic if it contains at least one parameter pack in the template parameter list.

Pack expansion When a template that has a parameter pack is used, the pieces of code using the pack in the template body undergo pack expansion, resulting in a specialization of the template. An example altered from [26]:

```cpp
template<typename... Us>
void g(Us... us);

template<typename... Ts>
void f1(Ts... ts){
    g(ts...); // "ts..." is a pack expansion, where "ts" is its pattern
}

f1(3, 3.0);
// Ts... ts is expanded to int E1, double E2
// ts...    is expanded to E1, E2
// Us... us is expanded to int E1, double E2
```

Template instantiation

C++ code containing only templates doesn't generate any executable code. In order for that to happen, the template needs to be instantiated, meaning that the template parameters are replaced with template arguments to create an actual class, function, or variable. An instantiation results in a so-called specialization of the template, which holds the class, function, or variable. Specializing an alias template has the same effect as substituting the template parameters of the class template that it aliases [28].

```cpp
template<typename T>
class V {
    void func(T t);
};
// Explicit specialization for T = int
template<>
class V<int> {
    void func(int t);
};
```

Such a specialization can either be explicitly performed by the programmer (allowing to adapt the definition if needed) or by the compiler if the programmer asks for an instantiation.

A template specialization is a specialized form of a template with some or all template parameters replaced by arguments. A template specialization is the class, function, or variable the user or the compiler generates, whereas the instantiation is the *process* of creating that specialization [3].

Explicit instantiation Explicit instantiation refers to explicitly asking the compiler to create such a specialization, e.g., like so in the case of class templates:

```cpp
template<typename T>
class V {};
// Explicit instantiation
template class V<int>;
```

Any explicit instantiation can only be requested once, else you will get compiler errors. Only class, function, and variable templates can be explicitly instantiated.

Implicit instantiation When at some point in code the definition of a specialization of a class, function, or variable is needed, and it has until that point not yet been explicitly instantiated or explicitly specialized, the compiler will perform an implicit instantiation [1].

```
template<typename T>
class V {};

// Code requiring an instantiation of V
V<int> v;

// Specialization of V generated by the compiler
// (compiler-generated, so not visible in the source code)
template<>
class V<int> {};
```

Variable templates

Variable templates are – despite the name – a means to create parameterized *constants* [34], and can only be declared at namespace scope or `static` at class scope [29].

```
template<typename T>
T data;
```

Before C++14 introduced variable templates, two idioms were typically used [29]:

- `static` data member of class template
- `constexpr` function template returning the desired value

```
template<typename T>
class Widget {
public:
    static T data;
};
```

Listing 2.3: `static` data member of class template

```
template<typename T>
constexpr T f1(){
    T data;
    return data;
}
```

Listing 2.4: `constexpr` function template

Variable templates can be used, for example, to create a function that does computations with varying precision (example slightly altered from [34]).

```
template<typename T>
T pi = T(3.1415926535897932385);

template<typename T>
T area_of_circle(T r){
    return pi<T> * r * r;
}
```

The idioms have a couple of drawbacks compared to variable templates, namely more confusing syntax and worse `constexpr` support. For more details, see [34].

2.2 C++ Standard Library

Apart from the fact that the C++ core language is not small by itself, it also boasts a huge Standard Library, not to speak of third-party libraries.

Containers The C++ Containers Library brings us data structures with many different flavors: sequence containers (arrays, lists, and the like), unordered and ordered associative containers (maps indexed by keys), and container adaptors (provide stack- or queue-like access to sequential data structures) [31]. The most important properties of a container are probably the type of container (is it a `std::vector`, is it a `std::unordered_multimap`?) and the type of contents of the container. Another property of interest could be the

"depth", i.e., the degree of nesting used at a container, such as a degree of 2 in `std::vector<std::map<std::string,int>>` for example.

Utilities The C++ Standard Library also comprises various utilities, including

- `std::pair` and `std::tuple`: data structures for pairing up two or a variable amount of different-type values,
- smart pointers like `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr` to ease memory management by defining ownership semantics.

Algorithm The Algorithms Library provides a wide range of function templates to perform computations on ranges of elements, like searching for an object, counting, sorting, or modifying the range. It seems that its function templates are rarely used, a presumption we would like to underpin or undermine with real-world data.

2.3 Deprecated and unsafe constructs, best practices and anti-patterns

LLVM's `clang-tidy` is a modular linter to help programmers write better code "by fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analsis" [11]. Unlike `clang-tidy`, this framework's goal is not to fix errors immediately, but to collect data to be able to assess *why* errors are made, enabling to take action at the teaching and language design levels to improve the quality of codebases.

In the following, we give a couple of examples of programming practices.

`override`, `final` `override` and `final` are both keywords occurring when programming in an object-oriented fashion in C++. Virtual functions are the mechanism that enables to call methods from a derived class through a pointer of base class type:

```
struct Base {
    virtual void f();
};
struct Derived : public Base {
    virtual void f();
}
Base d = new Derived();
d->f(); // Will call Derived::f
```

The derived class's method will be called because its `f` overrides the one from the base class. `Derived::f` overrides `Base::f` because the functions have the same signature. `override` is used to explicitly tell the compiler that the function declaration of a derived class is supposed to override the base class declaration. This helps to catch errors because the compiler will complain if the signatures are incompatible. If you erroneously mistype a function and omit the `override`, the compiler cannot mark that as an error because it has to assume that the derived class declares a new function.

The `final` keyword, in some sense, has the opposite meaning, namely that a class cannot be derived from or that a virtual member function cannot be overridden. Firstly, it can help to make the design of the program clearer. Secondly, by declaring things `final` you can help the compiler with devirtualization, possibly gaining performance increases by allowing the compiler to figure out which function to call at compile-time rather than at runtime [17].

The rule of three/five/zero Without going into too much detail, the rules of three, five and zero all stem from the fragility of C++ classes that handle heap memory, e.g., through a raw pointer. The compiler is your friend and tries to help by implicitly defining copy/move constructors and assignment operators for your classes if you don't do it or explicitly tell the compiler not to. However, if your classes manage external memory, the compiler-generated functions are usually incorrect, e.g., because the generated copy operations result in shallow copies of the managed memory. The rule of three/five/zero provides good hints as to when to define which member functions. The potential for analysis that I see is that we could check whether programmers leave performance on the table by suppressing the generation of move operations [32].

2.4 General-purpose features

Loop depth Another feature we can look for in code is loops. Loops have a big performance impact, making them interesting to investigate. I define the depth of a loop statement recursively as 1 + the maximal depth of all contained loop statements. The goal of this feature is to get a rough performance estimate by analyzing the prevalence of loop depths. Not distinguishing the different kinds of loop statements (`for`, `while`, `do-while`, range-based `for`), the example below has a depth of three:

```
for(int i=0; i<n; i++){
    while(b1)
        if(b2){
            foo();
        } else {
            bar();
            for(auto i : arr){
                baz();
            }
        }
}
```

2.5 Software metrics

Cyclomatic complexity The cyclomatic complexity computes the number of linearly independent paths through a procedure, or in C++ parlance, a function. In the *control flow graph* of a function, a node designates a block of code without any branches and an edge designates a branch as experienced at for example an `if`-statement or a loop condition. The cyclomatic complexity describes the number of paths through that graph s.t. each path has at least one edge that is not contained in any other path. Thomas J. McCabe criticized the practice of measuring code complexity by its physical size and developed this measure to find pieces of code with high complexity that need extra attention when testing software [39].

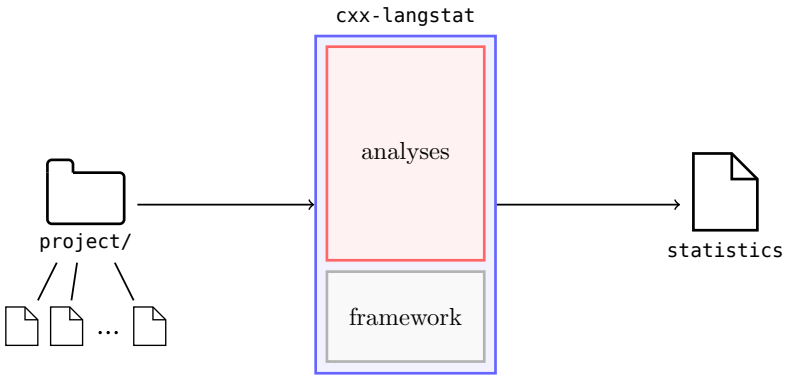
Chapter 3

Framework

`cxx-langstat` is a flexible framework based on Clang that allows arbitrary analyses to be run. I will now present how it works, what it consists of, and what its dependencies are.

3.1 Tool

The thesis aimed to create a tool for the analysis of C and C++ code. Given a C/C++ project, the tool should analyze that project by computing statistics about the adoption of language features or the prevalence of certain constructs and idioms. The tool is a basis for future analyses that we want to perform. This leads to a design where the tool consists of a framework and the analyses built on top of it. `cxx-langstat` already contains multiple analyses and is modular s.t. it can be easily extended with new analyses. Using a flag that accepts a comma-separated list of analyses we can specify what statistics to gather from a project.



Currently, the analyses are compiled into a single static library and then linked together with the framework to build `cxx-langstat`.

LibTooling To build a tool on top of Clang, I had to choose how to interface with the LLVM frontend. I finally settled on *LibTooling*, because it was advertised by LLVM as a support to build standalone tools based on Clang [8] and allows to use so-called *AST matchers* to match interesting nodes in the AST [9] (more information about ASTs and matchers in later sections).

In the following sections, I will explain the roles of the framework and the requirements it poses to analyses s.t. they can be run by it.

3.2 Framework

The framework is the basis of the tool where new analyses can be registered. It is an API that performs the gathering of statistics in two stages, in each of which it calls a function from an analysis:

1. Emitting features from code
2. Computing statistics from features

To give some intuition about the stages I will explain them using a hypothetical analysis whose functions will be called. Its goal is to get data about the distribution of the return types of functions in C++ code. Although this is a bit of a contrived example, this data could be useful for example to study the effectiveness of C++'s *return value optimization* (RVO). In the first stage, the objective would be to extract all function declarations along with their return types. Then, to compute statistics, the analysis would count for each return type how often it occurred, yielding us the desired data.

When we want to add a new analysis, we have to implement two functions that the framework relies on in the respective stages. Every analysis looks roughly as follows:

```
class Analysis {
    std::string Name;
    // input: C/C++ code
    // output: features
    getFeatures(){
        // implementation-dependent
    }
    // input: features
    // output: statistics
    getStatistics(){
        // implementation-dependent
    }
};
```

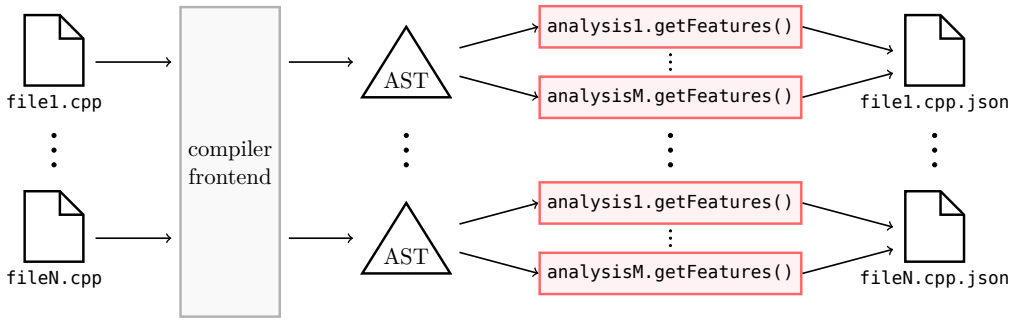
In the following, we give an overview of the stages of the tool and how they rely on the functions of the registered analyses. Details about computations performed by the analyses in each stage are deferred to later sections.

3.2.1 Stages of the framework

In this section, I will explain how the framework executes the two stages and how it relies on the analyses to succeed.

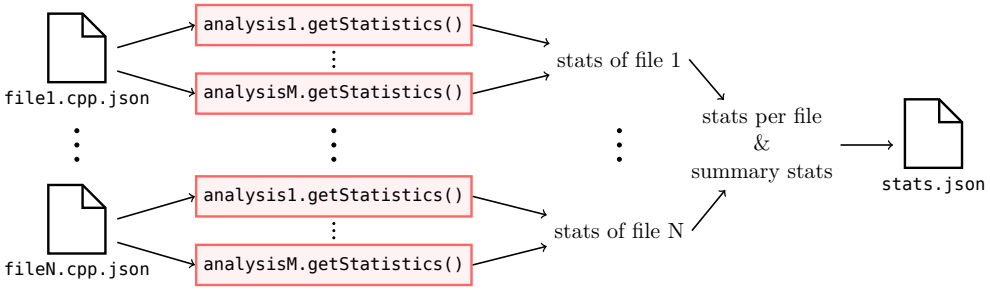
Emitting features from code When the tool is requested to emit features, the tool will recursively search the project directory for all source files (`.c`, `.cpp`, `.h`, etc.) and AST files (`.ast`) generated from source files. As we will describe in more detail later, this stage acts on the *abstract syntax tree*, or AST for short, of the code. For each input file, depending on whether it is a source file or an AST file, the tool will invoke the compiler frontend to either parse the source file to AST or load the AST from disk into memory. It might be necessary to provide so-called compile commands, either directly on the command line or via a compilation database, for the frontend to parse the input correctly [15]. The AST is now stored in memory as an `ASTContext` and can be passed to any analysis for further inspection. We call the `getFeatures` function of each analysis with the `ASTContext` as the argument, yielding for each input file a plethora of extracted features. These features are then, grouped by their input file, written to disk in `.json` files. Using a shorthand notation, we indicate the name of the analysis that created a feature. We chose the JSON (JavaScript Object Notation) format to store and write our features because of the

heterogeneity of the extracted features and the good readability of JSON. Also, using a JSON library[1] it is easy to create and parse `.json` files.



It should be noted that one should store the files containing the extracted in a separate directory s.t. we can be sure that in the next stage we don't accidentally try to compute statistics from non-features files otherwise lingering around. Because each input file is parsed and analyzed independently from all other inputs, this stage lends itself to be trivially parallelized (admittedly, the command line output is garbled, but that can be easily fixed once the standard library implementations implement C++20's `basic_osyncstream` – until then, we would have to resort to third-party libraries). A runner wrapped around this stage will, when asked to, create multiple batches of input files, which can then be analyzed in parallel. The parsing from source to AST and getting features from the AST are the most expensive routines of the program, thus parallelization is beneficial.

Emitting statistics from features In the second stage, the tool will instead look for `.json` files, given a directory. We pass each file containing features to the `getStatistics` function of every enabled analysis. Since the features are tagged with the name of the analysis that created them, we can ensure that each analysis only ever computes statistics from features that it created itself. This way, we have statistics for each source/AST file that we created features for. Finally, because often it is interesting to compute statistics about a whole project instead of a single file, we condense all of them into a summary statistic.



We store features in multiple files but statistics in a single file because the files containing the features tend to be larger.

The reason that we perform this two-step process and do not directly compute statistics from source code is that features are more human-readable, which aids in debugging. From statistics, it would be difficult to verify the correctness of the extracted data. Features can be directly compared to the source code they were extracted from because they can indicate their locations. Second, if we enhance our features with some extra data (e.g., for each extracted item also store other relevant information about it), it enables us to compute new statistics later, avoiding expensive reparsing of the source code and reextraction of features.

[1] https://github.com/nlohmann/json

3.3 Analysis internals

Now that we've seen how the framework executes in two stages the functions provided by the analyses, I want to detail the kind of computations the analyses perform.

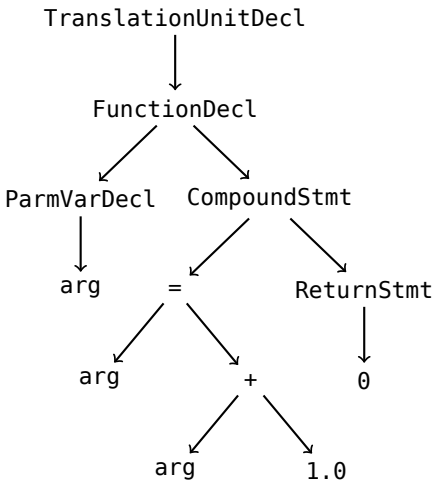
3.3.1 Emitting features from code

In the first stage, the tool's job is to orchestrate the analyses to emit features from the input code. In order to better understand how an analysis extracts features, recall its definition is from 2.1.

Abstract syntax trees To get a grip on the code we want to analyze, we leverage Clang's power and use it as a library to retrieve the so-called *abstract syntax tree* representing the code. An abstract syntax tree, or AST, is the result of the parser: after lexing, when all tokens of the source code have been determined, the parser's job is to traverse the token stream and to build a tree representing the abstract syntax of the code [44].

```
int func(double arg){
    arg = arg + 1.0;
    return 0;
}
```

The C++ code intuitively corresponds to the tree on the right representing its syntax.



The top node, the `TranslationUnitDecl`, can be thought of as storing all relevant syntactical information about a single source file of which all C/C++ preprocessor actions have already been performed, like the inclusion of headers [4] (p. 10). The `FunctionDecl`, together with the `ParmVarDecl`, stores the signature of the function, i.e., the parameter and return types of the function. The `CompoundStmt` should intuitively be thought of as its body or the definition [7].

In more detail, the Clang library stores the tree as the aforementioned `ASTContext`, see listing 3.1. Let us call it the Clang AST for short from now on. The Clang AST contains apart from declarations, statements, and expressions also the types, the line- and column numbers, and the memory addresses of the AST nodes.

Extracting relevant code locations In order to be able to extract useful locations from the AST, we use *LibASTMatchers* [9, 10], a Clang library to "match" nodes in the AST which helps us to retrieve pointers to nodes in the AST that satisfy a certain property. LibASTMatchers provides a domain-specific language that we use to build predicates. An example of a matcher would be

```
functionDecl(
    has(CompoundStmt().bind("body")))
.bind("function")
```

The matcher above will match function declarations that have a body. Given a match, the `bind` commands can be used to retrieve pointers to the respective nodes that they bind to.

```
TranslationUnitDecl 0x7fd4c2026c08 <<invalid sloc>> <invalid sloc>
-FunctionDecl 0x7fd4c2055098 <ast1.cpp:1:1, line:4:1> line:1:5 func 'int (double
    )'
  |-ParmVarDecl 0x7fd4c2054fc0 <col:10, col:17> col:17 used arg 'double'
  `-CompoundStmt 0x7fd4c2055270 <col:21, line:4:1>
    |-BinaryOperator 0x7fd4c2055220 <line:2:5, col:15> 'double' lvalue '='
    | |-DeclRefExpr 0x7fd4c2055188 <col:5> 'double' lvalue ParmVar 0x7fd4c2054fc
        0 'arg' 'double'
    | `-BinaryOperator 0x7fd4c2055200 <col:11, col:15> 'double' '+'
    |   |-ImplicitCastExpr 0x7fd4c20551e8 <col:11> 'double' <LValueToRValue>
    |   | `-DeclRefExpr 0x7fd4c20551a8 <col:11> 'double' lvalue ParmVar 0x7fd4c
        2054fc0 'arg' 'double'
    |   `-FloatingLiteral 0x7fd4c20551c8 <col:15> 'double' 1.000000e+00
    `-ReturnStmt 0x7fd4c2055260 <line:3:5, col:12>
      `-IntegerLiteral 0x7fd4c2055240 <col:12> 'int' 0
```

Listing 3.1: Clang AST as seen using `clang++ -XClang -ast-dump`

Even though nodes are often related in the AST, having multiple `bind`s is often useful if you need pointers to multiple nodes which are otherwise more laborious to get, e.g., when the tree is very deep.

Because most C/C++ files include headers, the `isExpansionInMainFile` matcher was virtually present in every matcher that I built. It ensures that nodes matched are written in the currently analyzed file and do not come from an included header file.

Clang provides a tool called `clang-query` for prototyping matchers, which can later be embedded into your C++ source code [38, 10]. For an extensive list of available matchers, see [5].

Extracting relevant data from code locations Note that according to the definition, we've already extracted features. Having the relevant code locations can be enough – some analyses that only do simple counting would fare great with only knowing the code locations, e.g., reporting just the function declarations' existence would suffice for an analysis that wants to count how many function declarations there are.

```
{
    "function1": {
        "location" : 6
    },
    "function2": {
        "location" : 13
    }
}
```

Listing 3.2: Features containing only information about functions' locations

Imagine however a hypothetical analysis that additionally would want to gain insights about the return types of functions and report the most popular ones. In order to do that, we again leverage the Clang library: for each `FunctionDecl` we can call `getReturnType()`. The JSON file containing the features would then look like this:

```
{
    "function1": {
        "location": 6,
        "ReturnType": "int"
    },
    "function2": {
        "location": 13,
        "ReturnType": "double"
    }
}
```

Listing 3.3: Features also holding data about return type of functions

Analyzing code on this high, almost source-like, AST level and using Clang as a library means that we know as much about the code as the compiler – at the cost of having to get familiar with interfacing with Clang and being dependent on its development. We also conclude that how much we depend on the Clang library can differ a lot from analysis to analysis, depending on what features that we need to extract and what other data we need to get to be able to compute the desired statistics.

3.3.2 Emitting statistics from features

From features to statistics In abstract terms, computing statistics from features is applying a function to features. That function could compute for example:

- the absolute prevalance of a feature,
- the relative prevalence of features by comparing their commonness, e.g. to assess their popularities,
- the distribution of a variable contained in the features.

Recall our hypothetical analysis computing statistics about return types used and the features returned by it, see listing 3.3.

The analysis would then hold a map indexed by return type containing a value describing how often a return type occurred. By iterating over the JSON features, we can then easily compute for the file that these features were generated from the distribution of return types and write the results to a new JSON file like so:

```
{
    "Return Type Distribution": {
        "int": 1,
        "double": 1,
    }
}
```

Summarizing statistics Intuitively, if statistic X and statistic Y both contain data about some property a, then we take the "sum" of the statistics. If only X contains data about a, then it is assumed that Y's data about a is 0, thus the statistics from X are simply copied over to the summary. This is achieved using a simple recursive piece of code, which can be best laid out as follows:

Algorithm 1: Summarizes two statistics into one

Input: JSON objects X, Y both containing statistics;
Output: summary statistics S;
foreach *key pair* (k_x, k_y), $k_x \in X, k_y \in Y$ **do**
  v_x, v_y are the JSON values belonging to k_x, k_y, respectively;
  if $k_x == k_y$ **then**
    if v_x, v_y *are both JSON objects* **then**
      call this algorithm on v_x, v_y;
      store the result in S under key k_x;
    if *vx, vy are numbers* **then**
      store $v_x + v_y$ in S under key k_x;
  else
    store v_x in S under key k_x;
    store v_y in S under key k_y;

3.4 Interfacing with Clang

In this section, I will briefly describe the components/classes used to interface with Clang to get the Clang AST as required in the stage where we emit features. The three most important components are the `ASTFrontendAction`, `ASTConsumer` and `MatchCallback`. Peter Goldsborough presents an example in [36]. Also see [6] for information about frontend actions and consumers. When computing statistics from features we don't interface with Clang at all.

`ASTFrontendAction` This component can be subclassed to create your own actions that should be run by the frontend. We need the frontend only to generate the Clang AST from a source code file or read the AST from a `.ast` file and to create the `ASTConsumer`.

`ASTConsumer` Execution is then continued in the `ASTConsumer`, which contains a method that is entered as soon as the AST is ready. Its job is to create `cxx-langstat`'s analyses and run them s.t. they extract features from code for us. When for the current AST every enabled analysis has computed its features, it stores them to disk.

`MatchCallback` When you register a matcher to match nodes in the Clang AST, the so-called `MatchCallback` is called on a match. I have abstracted this callback to a functional interface – the framework provides to the analyses a function that accepts a matcher as an argument and returns the nodes in the Clang AST.

Using a *`ClangTool`* we tell the frontend to run our frontend action over our input files, possibly with a set of compilation commands for the frontend to be able to correctly parse our source code.

Chapter 4

Analyses

Definition 4.1 (Analysis) *The goal of an analysis is to extract one or more features and apply a function to them to compute statistics.*

4.1 Implemented analyses

In the following, I will describe the analyses that the tool already contains by showing for each analysis how it extracts features and what statistics it computes from them. Also, I will explain their limitations and propose improvements.

4.1.1 Cyclomatic Complexity Analysis (CCA)

Features The Cyclomatic Complexity Analysis first uses the `functionDecl` matcher to extract all the function and member function declarations. This includes function templates as well as their specializations and specializations caused by instantiations. All declarations that don't have a body are ignored since the cyclomatic complexity of a function is the number of linearly independent paths through its body. For each function, the control flow graph (CFG) is computed from which we calculate its number of nodes and edges. To get the cyclomatic complexity of a function, we compute $CYC = E - N + 2$, where E and N indicate the number of edges and nodes, respectively (see [39] for details). The intraprocedural CFG we can compute using Clang is suitable to apply McCabe's method to since it has a distinct entry and exit node. Finally, we store a list of names of function declarations paired with their cyclomatic complexities.

[36] provided an example implementation that was helpful.

Statistics By iterating over the previously created list, we get an indication of the commonness of the cyclomatic complexities in the source code.

Limitations If a function template contains a templated class type that is iterated over with a range-based `for` loop, CCA fails due to not being able to compute the CFG. In that case, the function template will be skipped and not reported.

```
template<typename T>
void foo(std::vector<T> vec){
    for(auto el : vec){ }
}
```

4.1.2 `constexpr` Analysis (CEA)

To analyze the adoption of the `constexpr` keyword, this analysis measures the percentage of constructs like variables, functions and `if`-statements declared `constexpr`.

Features CEA extracts four features:

Variable declarations After extracting all variable declarations, the analysis creates a list containing them and stores for each one whether it was declared `constexpr` or not. Additionally, it registers the identifier, type, and location of the variable. Some conditions that trivially make a variable not eligible to be `constexpr` are checked, meaning it is not reported if any of the following hold (see [30]):

- variable is a function parameter
- variable occurs in a structured binding
- variable that has no initializer

However, there are more necessary conditions for `constexpr` which are not checked, like whether the initialization of the variable can be determined at compile-time [30]. Also, another important limitation is that loop variables are reported as not `constexpr`. Loop variables should, however, never be reported in this analysis since those are typically incremented or changed, requiring them to be non-`const` and thus non-`constexpr`. Detecting this would require checking if a variable is a loop variable.

Function and method declarations Similarly to variables, it extracts all function and method declarations and stores whether they were `constexpr`, together with their identifiers, signatures, and locations. Again, there are many requirements for a function to be able to be `constexpr` which are not checked, even separate ones for class constructors and destructors [30].

`if`-statements For each `if`-statement we report its location and `constexpr`-ness. Here we should also check its eligibility for `constexpr` by looking if the value of the condition can be established at compile-time [22]. This is, as of now, not implemented.

Statistics For each of the four constructs that can be `constexpr` we report how many are `constexpr` and how many are not.

Limitations As is clear by now, this analysis has a big potential for improvements. Right now, it will indicate a low usage of `constexpr` due to it reporting constructs that cannot be `constepxr` anyway, therefore underestimating the real popularity of the keyword. The conditions for `constexpr` have changed significantly in C++14 and C++20 [30], making checking them more difficult to implement. To conclude, CEA currently gives us only insights into how often this keyword was used and not used, respectively; what we really want to know instead is how frequently it is used compared to how often it could have been.

4.1.3 Loop Depth Analysis (LDA)

This is the first and simplest analysis that I wrote which helped me to gain familiarity with C++ and Clang as a library. Given a maximal depth, LDA creates statistics on the commonness of different loop depths by reporting for each loop depth the locations of the loops with that depth, allowing us to get a very rough estimate of the complexity of the program. `for`, `while`, `do-while` as well as range-based `for` loops are considered, either when inside of a function or a class method.

Features The matcher to find a loop of depth d is conceptually very simple:

1. First, find all loops of depth at least d by nesting the matchers `loopStmt` and `hasDescendant` $d-1$ times:
   `loopStmt(hasDescendant(loopStmt(...hasDescendant(loopStmt()))))`

The `hasDescendant(p)` matcher is a Clang-provided matcher that matches if the current node has some node in its subtree that satisfies `p`. Furthermore, `loopStmt` is a custom matcher that matches any of the four kinds of loop statements.
2. Second, find all loops that are no deeper than d.
3. Take all loops satisfiying both of the properties above.

By applying this for every depth until the maximal depth, we can compute features containing for each depth the locations of the loop statements having that depth.

Statistics We count for each loop depth in the features how often it occurred.

Limitations

Exponential implementation Currently, the matchers grow exponentially in size with the maximal depth given due to the exponential number of combinations that the different loop statements can be nested. Even though in practice when analyzing real-world code this shouldn't be too much of a problem because loop depth will rarely be bigger than five or ten, it would still be reassuring to reimplement this for example using dominator trees. These are common in compilers and thus a means we could use to reimplement this.

Loop depths across function calls Imagine the following scenario:

```
void foo(){
    for(int i=0; i<n; i++){
    }
}
for(int i=0; i<n; i++){
    foo();
}
```

Are the two loops of depth one or one loop of depth two? If one wanted to use LDA to have a very rough approximation of program complexity, I think it could be a sensible addition to change the analysis to also look for loops that are "hidden" behind function calls.

4.1.4 Loop Kind Analysis (LKA)

To investigate whether programmers switched from the traditional loops to the new syntactic-sugar range-based `for` loop by counting the prevalence of each kind of loop. Naturally, this analysis thus extracts the uses of all the range-based `for` loops, but also the uses of the other kinds of loops to be able to compare them.

Features LKA extracts four features, namely for each of the four loop kinds, it extracts all locations of that loop kind. Luckily, LibASTMatchers provides for each loop statement kind a matcher, namely `forStmt()`, `whileStmt()`, `doStmt()` (do-while loops) and `cxxForRangeStmt()`.

Statistics Similar to other analyses, we can establish their commonness in the code by counting the number of locations that they occur at.

Limitations I alluded in chapter 2 to the fact that not all "traditional" loop statements can be converted to use the new syntax, like infinite loops and those that invalidate iterators during the iteration. This entails that the above usage statistics will not give us perfect insights into the adoption of the range-based `for` loop. We could build more complicated matchers and perform checks to filter out ineligible loops to get more indicative

data. This would allow us to find out how many range-`for` weren't used because it wasn't possible and how many weren't used because the programmer didn't adopt it.

4.1.5 Function Parameter Analysis (FPA)

The addition of rvalue references and move semantics motivate another analysis, namely the *Function Parameter Analysis*. To get first insights whether rvalue references and the move semantics it enables are used, we can analyze the parameters of functions and function templates.

I would like to first write down the notion of parameter kind: it describes how one can pass data to functions, namely by

- value,
- non-`const` lvalue reference,
- `const` lvalue reference,
- rvalue reference,
- forwarding reference.

Features FPA emits two features, one describing the functions, and a second describing their parameters.

- For each function, function template, or function template specialization (can be caused by instantiation), report function identifier, signature, and line number for debugging and for each parameter kind how often it occurred in the function parameter list.
- For each parameter, report its identifier and line number for debugging as well as its kind, its type, and if the type is dependent on the instantiation (meaning that the encompassing function is a template).

To get the features described, FPA proceeds as follows: First, it collects all functions and function templates; this includes member functions/methods but excludes constructors and assignment operators. I deemed it was a better idea to analyze those separately, e.g., in a future analysis that checks for compliance with the rule of three/five/zero [32]. Second, we iterate over those collected items and get their function parameters. For each parameter, we try to determine its kind by looking at its type. If its type is a pack we "unpack" it and look at the type that is being packed. In case that it is an rvalue reference type and the encompassing function is a template we additionally check if it could be a forwarding reference. If it isn't a reference at all, we determine it to be a by-value parameter. During this process, we also keep track of the parameters found so far and store them coupled with type and instantiation-dependency info.

Statistics FPA has two functions to compute statistics:

- `FunctionsCount`: For each parameter kind, it counts how many functions use them. We report this separately for functions and function templates because forwarding references can only occur in templates.
- `ParamsCount`: Count for each parameter kind how often it occurred. This is not just the sum of the above statistics, because a certain kind of parameter can occur multiple times in a single function's parameter list.

Limitations

Function parameter packs For function parameters that are parameter packs, TPA currently doesn't report the fact that they are parameter packs. Although it is reported that they are instantiation-dependent (as parameter packs inherently are), we thus miss out on the opportunity of analyzing if function parameter packs are common or not.

Separate functions and parameters Currently, functions and parameters are reported separately, thus making it hard to backtrack which parameter belonged to which function. As of now, I don't have any striking ideas that would necessitate that, but it might inhibit the computation of interesting statistics in the future.

4.1.6 Template Instantiation Analysis (TIA)

This is the most complicated analysis so far. The *Template Instantiation Analysis*, or TIA for short, analyzes which template specializations are used how often. Its name comes from the fact that it analyzes the usage of template specializations that were caused by an instantiation of a template. Chen et al. [18] point out that templates can be used by instantiation or derivation – in this analysis, I however only study template usage through their instantiations and the arguments used to create the instantiations.

TIA regards three flavors of templates, namely classes, functions, and variables. Alias templates are indirectly studied by reporting their instantiations as instantiations of the class that they alias. Because we don't want to use TIA directly, but instead use it as a basis for more specialized analyses s.t. we can analyze different templates separately, TIA has three parameters that can be used to control its execution:

- `TemplateKind`: this parameter determines which kind of template instantiations we want to analyze, i.e. whether class, function, variable, or all three kinds of templates should be respected.
- `Names`: a list of names s.t. a template's instantiation is only reported if the name of the template comes from `Names`.
- `HeaderRegex`: this regex is to be used in conjunction with the `Names` parameter: because the name of a template (even when fully qualified) does not uniquely determine the template itself, we can additionally constrain which instantiations are interesting to us by using `HeaderRegex` to condition on the header where the template was defined. The initial motivation was because of the `std::move` function template in the C++ Standard Library, which occurs twice in different contexts. It is crucial to note, when using this regex, that it has to specify the header where the definition of the template lives and not the header which was used to get access/include the template. A notable example is again `std::move`, which according to the C++ Standard is accessible once you include `utility`, but which is implemented in a different headers (e.g. in `type_traits` in LLVM's `libc++`[1] and in `bits/move.h` in GNU's `libstdc++`[2]).

Features

TIA can report features in two ways, depending on the feature:

- Binary: report an instantiation only once if and only if it exists
- Countable: report every use of an instantiation, allowing to count them and thus compare commonness and assess the popularity

For the three kinds of templates that TIA regards, it extracts instantiations or uses of specializations created by instantiation and stores the template arguments that were used to create those specializations.

Class templates

[1] https://github.com/llvm/llvm-project/blob/release/11.x/libcxx/include/type_traits#L2613
[2] https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-api-4.5/a00936.html

Explicit class template instantiations Explicit class template instantiations are the easiest to describe:

```
template class V<int>;
```

Explicit instantiations are used to explicitly request a specialization from the compiler. We match them by looking for class template specializations that have been created as part of an instantiation. I've noticed that in the Clang AST a specialization that was explicitly instantiated is not stored in the subtree of the class template, but just next of it as a "sibling" subtree – this was the criterion used to distinguish explicitly instantiated from implicitly instantiated specializations. TIA reports them too just because this feature was added early on, and only later I realized it wasn't that important – but why remove it? This is a binary feature; it cannot be counted because any explicit instantiation can only be issued once. For each explicit instantiation detected, it will report the template arguments used to create the specialization.

Implicit class template instantiations In this paragraph, I will describe how I analyze code that either

- causes an implicit class template instantiation (because it is the first time the definition of the specialization is needed),
- uses a type that was earlier implicitly instantiated (an earlier piece of code implicitly instantiated it) or
- uses a type that was earlier explicitly instantiated (the programmer explicitly instantiated it earlier).

Illustration:

```
// Template
template<typename T>
class V {};

V<int> v; // Causes implicit instantiation, definition of V<int> first needed
V<int> v2; // Uses type that was implicitly instantiated earlier

template class V<double>; // Explicit instantiation (not analyzed in this
    paragraph)
V<double> v3; // Uses type that was explicitly instantiated earlier
```

I realize that this is a broad range, but I just need some term to distinguish these cases from explicit instantiation as in the previous paragraph. Calling all those cases "implicit instantiation" is a stretch of the terminology, but it makes sense to bundle these cases since they are syntactically similar. The goal really is to study which specializations of a template are more popular and which are less popular.

As a rule of thumb, let us consider in this part all instantiations "implicit", if they occur in C++ code through "natural" usage, i.e. anything that is not an explicit instantiation request to the compiler.

- Variable and member variable declarations:
  The first scenario that comes to mind when you want to study the use of class template instantiations is (member) variable declarations. Not every variable declaration of template specialization type causes an implicit instantiation, namely because it can have already previously explicit instantiated, implicit instantiated by another variable, or explicitly specialized. Our goal here is to study variable declarations that have a type that is a specialization of a class template, because it gives us insights into how class templates are used.

```
class Widget {
    V<int> mv; // member variable
};
V<int> v; // variable
```

Listing 4.1: Variables requiring an instantiation

The matcher to extract such code is the following: it looks for all (member) variables whose canonical type is a specialization of a class template, where the specialization has to be a result of an instantiation.
The canonical type requirement is important because it allows us to look through all the type definitions and aliases and enables us to report variables that use type synonyms to be reported as a use of the class specialization their type aliases.
As we look at each variable having a suitable type dictated by the matcher, we can use this to generate statistics on how popular the different specializations are. This is useful e.g. to assess the contents of STL containers, like how often `std::vector<int>` is used compared to `std::vector<unsigned>`. These reported specializations are interesting because they allow us to gain extensive insights into how library types and user-generated types are used in code.

Since function parameter declarations are also variable declarations, function parameters will also be reported.

Am improvement would be to also report variables that are references or pointers to a class template specialization (e.g. instances of `std::vector<int>*` or `std::vector<int>&`). I was unsure whether to include those variables: first, because a pointer to a class specialization does not cause an instantiation because the complete type might not required [20]. Second, if we match references to class specialization, are we then also interested in parameter variables that would match from the copy and move constructor?

To conclude, we report each variable/member variable/function parameter declaration including its line number in the code and the arguments that were to instantiate the specialization of the template. Because every use in a variable declaration is reported and multiple variables can use the same specialization, this is a countable feature.

- Template specialization type template type arguments:
  I know this is a mouth full. Let me explain: as we're talking about about template instantiations, "template specialization type" means "type that is a template specialization", e.g. `std::vector<int>` is a type that is a specialization of `std::vector`. A "template type argument" is a template argument that happens to be a type, thus the `int` in `std::vector<int>`. A limitation of TIA is that it will not report template type arguments that are template specialization types. Here's an example to illustrate why:

  ```
  template<typename T>
  class Widget {};
  Widget<Widget<int>> w;
  ```

  Listing 4.2: Nested instantiation

  Variable `w`'s type is a specialization of `Widget`. Great, so it will be reported and added to the features. However, the inner `Widget<int>` specialization will never be reported because it is neither explicit nor is it the type of a variable declaration. There are more cases of uses of implicit class instantiations that are not detected because they don't fit into the variable declaration category, e.g. literals or return types of functions.

Function templates Similar to class templates, you can either explicitly instantiate function templates with a similar syntax to the one for class templates, or you have implicit instantiations and pieces of code using them.

Explicit function template instantiations TIA does not report any explicit function specializations due to time constraints.

Implicit function template instantiations Similar to class templates, I report pieces of code that cause implicit instantiations, use a function that was implicitly instantiated earlier, or use a function that was explicitly instantiated earlier. This was motivated by the Algorithm Library Analysis, which I will lay out later.

- Call expressions:
  This part of TIA reports every function call to an instantiated specialization of a function template. Because we report every call to a function template, we can again assess which function instantiations are the most popular. A limitation of this is that we only analyze function templates' here through call expressions, i.e., we don't know how function templates are used in function pointers. I deemed this however to be acceptable, since I expected function calls to outnumber function pointers, hoping it to skew the final features and statistics not too much.

  Finally, we report each call expression to an instantiated specialization of a function template together with the line number of the call in the code and the template arguments of the specialization. This is also a countable feature.

  Member call expressions:
  TIA analyzes both functions and member functions. Member functions of class templates are a bit of a quirk: although they are templated, there aren't actually any template arguments to report, as they are completely dependent on how the class template encompassing them is instantiated. For that reason, I've decided to not report them as function template instantiations [27] (see templated entities).

```
template<typename T>
class C {
    void f(){}
};
```

Listing 4.3: Member function of class template

  Of course, if the member function is a template (i.e., it has the `template` keyword) I will report it including the template arguments.

Variable templates In contrast, this part of TIA is binary. Instead, it goes through all variable template specializations that result from an instantiation, no matter if explicit or implicit. It simply reports the fact that a certain instantiation of a variable template exists, but does not report every use of that instantiation if it were multiple. Variable templates are already a rather obscure construct, so I decided to keep it simple. We report thus for each variable template every instantiated specialization once incl. the instantiation arguments.

Statistics

TIA provides two functions for computing statistics from features:

- `templatePrevalence`: Given features describing instantiations, this function will compute numbers on which templates were used how often. It does not make sense to apply this to binary features since in that case, the result will be 1 for every template instantiation that occurred.
- `templateTypeArgPrevalence`: Given features describing instantiations, this function will compute numbers on how often a template instantiation was used. It will do that by counting for each set of template *type* arguments how often it occurred. One

could certainly also do this for template non-type and template template arguments. Again, as above, it doesn't make sense to apply this function to binary features.

For example, from the features extracted from below code, by applying above functions we would get the following statistics:

```
std::vector<int> v1;
std::vector<int> v2;
std::array<42, unsigned> a1;
```

```
// From templatePrevalence:
"container type prevalence": {
    "std::vector": 2,
    "std::array": 1,
},
// From templateTypeArgPrevalence
"container instantiation type arguments": {
    "std::vector": {
        "int": 2
    },
    "std::array": {
        "unsigned": 1
    }
}
```

These kinds of statistics can help us identify which containers or algorithms are popular and what they frequently contain.

Limitations

Implicit class template instantiations We see that there are still many cases left where we don't detect instantiations because they don't fit into any of the categories. If I were to implement this part now, I would rather use the `typeLoc` matcher, which allows extracting locations where a certain type is used. Instead of having to come up with cases where an interesting type can crop up beforehand, using `typeLoc` has the advantage that it yields directly those locations and we then can decide whether those interest us or not. A quickly prototyped matcher that I think is useful:

```
typeLoc(isExpansionInMainFile(), loc(hasDeclaration(
    classTemplateSpecializationDecl())))
```

After all, our primary interest is the usage of a type, so it makes sense to start with that and only then to zoom in. These limitations also apply to function and variable templates, but I think for class templates they are the most glaring.

TIA is not so much an analysis to be used directly because its results can be somewhat overwhelming due to the large number of instantiations reported. The main motivation for TIA is to instrument it to build more specialized analyses that I hinted at earlier.

Container Library Analysis (CLA)

This analysis aims to analyze the use C++ Standard Library containers. Since those containers are types, CLA instruments TIA to only study class templates. Moreover, as the argument to `Names`, we pass the names of all the containers we're interested in. In our case, that are all sequence containers, (unordered) associative containers, and container adaptors. For a full list of all containers, see the appendix.

Features To get the relevant features about containers, CLA subclasses TIA to create a constrained version that only analyzes instantiations of class templates using the `TemplateKind` parameter. Example:

```
#include<vector>

template<typename T>
class C {
    std::vector<int> mv;
};
int f(std::vector<double> dv);

int a =
    f(std::vector<double>{});
```

Notice that CLA detects the `std::vector<double>` parameter variable in the declaration of `f` and the `std::vector<int>` even though the encompassing template `C` is not instantiated, but not the `std::vector<double>` literal since it is not a variable declaration.

```
"cla": {
    "explicit class insts": null,
    "implicit class insts": {
        "std::vector": [
            {
                "location": 5,
                "arguments": {
                    "non-type": [],
                    "type": [
                        "int",
                        "std::allocator<int>"
                    ],
                    "template": []
                }
            },
            {
                "location": 7,
                "arguments": {
                    "non-type": [],
                    "type": [
                        "double",
                        "std::allocator<double>"
                    ],
                    "template": []
                }
            }
        ]
    }
}
```

Statistics To assess container popularity and prevalence of container contents, CLA can just call TIA's `templatePrevalence` and `templateTypeArgPrevalence`, respectively.

Utility Library Analysis (ULA)

ULA goes about it just like CLA, with the only difference that it analyzes a different set of class templates' usage:

- `std::pair`
- `std::tuple`
- `std::bitset`
- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

Notice how these are not all of the C++ General utilities library provides, but (probably, upon further analysis) the most important ones. One should be able to easily extend this to encompass more utilities.

Algorithm Library Analysis (ALA)

Features Currently, ALA subclasses TIA to analyze calls to function templates. Example:

```
#include <algorithm>

int main(int argc, char** argv){
    int a = std::min(3.0, 4.0);
    return 0;
}
```

The compiler automatically deduces from the arguments the type to instantiate `std::min`.

```
"ala": {
    "func insts": {
        "std::min": [
            {
                "location": 4,
                "arguments": {
                    "non-type": [],
                    "type": [
                        "double"
                    ],
                    "template": []
                }
            }
        ]
    }
}
```

ALA does not analyze all C++ Standard Library Algorithms, see the appendix for a full list.

Statistics ALA, right now, uses only `templatePrevalence` to assess the popularity of the different function templates.

4.1.7 Move Semantics Analysis (MSA)

FPA already gives some insights into the adoption of move semantics by analyzing the prevalence of rvalue and forwarding references, but since pass-by-value function parameters can also trigger move semantics, I decided to add an analysis that checks two things:

- Finds all calls of `std::move`, `std::forward` (Part 1).
- For functions that take a parameter by value, we check calls to those functions and try to analyze if those parameters are being copy- or move-constructed (Part 2).

Part 1 is fairly simple since we can instrument TIA to solve this problem. Part 2 needs some more attention: it achieves its goal via the following matcher: the matcher looks for all call expressions to functions with by-value parameters where the corresponding argument is a `CXXConstructExpr`, which is the node in the AST corresponding to the call of a constructor[3]. We can then leverage the Clang library to check if that construction is a copy or move construction.

Features

`std::move`, `std::forward` prevalence To get a first idea about the adoption of move semantics, we analyze whether the to move semantics two integral function templates are used: `std::move` and `std::forward`. Here again, TIA comes into play: as expected, I constrain it to look for calls to specializations of the above templates, keeping in mind to specify the correct `HeaderRegex` to not report the wrong moves (unfortunately, the C++ Standard Library contains two templates with the name `std::move`).

Construction of pass-by-value function parameters To balance out FPA's weakness with regards to pass-by-value parameters (from a pass-by-value function parameter we cannot determine whether a copy or a move occurs), I analyze at the call site whether the parameter will be copy- or move-constructed. To do that, conceptually, I match the call

[3] https://clang.llvm.org/doxygen/classclang_1_1CXXConstructExpr.html

expressions that, for each by-value parameter of the function that they call, have an expression that constructs the parameter. Note that functions include member functions and call expressions include calls to member functions. In this example, `c` on the last line is an expression (a `CXXConstructExpr`) that constructs the parameter `c_prime`.

```
class C {
    ...
};
void func(C c_prime){
    ...
}
...
C c;
func(c);
```

Using the Clang library, we can find out what constructor was used to initialize that parameter, allowing us to determine if a copy or a move took place. Also, we can check if the constructor used was written by the programmer or generated by the compiler, which can be useful to find out if the move was due to increased awareness of programmers about move semantics.

Subtleties:

- Copy elision: When a construction is *elidable* the question is whether to report those cases as copy/move or not. Currently, if we can determine from the `CXXConstructExpr` the constructor that was used to initialize the parameter, we report the copy/move, no matter if it might be elided in the end. If we cannot determine which constructor was used, this usually hints at the construction having been *elided*. We then report the construction as "unknown", ensuring that in any final statistic it will not be counted as a copy or move.
  The results can vary depending on the version of C++ due to changes in the specification of copy elision in C++17 [21].
- `CXXBindTemporaryExpr`: When the `CXXConstructExpr` constructs a class with a nontrivial destructor (i.e. it is user-defined, most likely because the class manages external memory which needs to be `delete`'d), the `CXXConstructExpr` has a `CXXBindTemporaryExpr` parent node in the Clang AST. The Clang documentation specifies that a `CXXBindTemporaryExpr` ensures that the destructor for some temporary is called[4]. I'm not sure why this is needed – it confuses me since that node is even part of the AST when a by-value function is called with an lvalue argument, meaning there is no temporary to be destructed. I have, however, changed the matcher to account for this because there are many classes with nontrivial destructors whose copy/move behavior we don't want to miss to analyze.

To conclude, this part of MSA will report the call expression together with information about whether the function will receive the parameter by copy or move, whether the copy/move constructor leveraged was compiler-generated or not, and the type of the argument.

Statistics We currently compute the following statistics:

- Number of uses of `std::move`, `std::forward`
- Number of copy/move constructions of by-value function parameters at call sites:
  - per type
  - in total

[4] https://clang.llvm.org/doxygen/classclang_1_1CXXBindTemporaryExpr.html#details

Limitations

Copy elision It might be advisable to explicitly state when copy/move is elidable or elided. Also, we need to be able to detect if the copy/move is guaranteed to be elided, which can be achieved by checking if the `CXXConstructExpr` is a `CXXTemporaryObjectExpr`.

Returning by value The same analysis of the construction could (should) also be performed when returning by value, possibly in liaison with an analysis measuring how many functions return by value.

4.1.8 Template Parameter Analysis (TPA)

The *Template Parameter Analysis*, or TPA, analyzes for each template (classes, functions, variables, and aliases) properties about their template parameters. The main motivation for it was to find out how often template parameter packs were used.

Features TPA operates as follows:

1. Get pointers to all templates used in the code – no matter the kind of template, but store them in four different vectors. This achieved using the Clang matchers library.
2. Iterate over all templates found, and count for each kind of template parameter how many it contains.
3. Iterate over all templates and set a flag indicating if it uses a template parameter pack. Note that this just analyzes template parameter packs, not function parameter packs – after all, we're analyzing template parameters here.

For example, the features for class templates look like this, indicating that there is some template `Widget1` that has one non-type and one type parameters, and that does not use parameter packs.

```
"class templates": {
    "Widget1": [
        {
            "location": 10,
            "parameters": {
                "non-type": 1,
                "template": 0,
                "type": 1
            },
            "uses param pack": false
        },
        ...
    ],
    ...
}
```

Note how the JSON value corresponding to the key is a list. This is because template names can be overloaded, notably functions.

Statistics As of writing the thesis, I only compute a single statistic from the above features: for each kind of template, I report how many templates use parameter packs and how many don't.

Limitations

Better reporting of parameters Instead of directly counting the different kinds of parameters, extract more detailed information about them, for example for non-type parameters what type of argument is required. For template parameters, we could even recursively employ TPA to get more information about the parameters of template template parameters. The fact that it starts counting already in the feature-extracting stage is an artifact from when the features-statistics distinction wasn't yet as clear.

Better reporting of packs Instead of a boolean indicating if a pack was used or not, reporting how many packs a template uses could be useful to investigate if e.g. a small number of templates contain a disproportionately large amount of all parameter packs.

Specializations Template specializations (of classes, functions, or variables) and partial template specializations (of classes or variables) are not analyzed for their template parameters because the matcher only looks for the primary templates.

4.1.9 `using` Analysis (UA)

This analysis aims to analyze the usage of type definitions and type aliases, or for short, (type) synonyms. Note that this analysis is called `using` analysis due to the main motivtion of analyzing if `typedef` was dropped in favor of `using` – it does not analyze the use of `using`-declarations like "`using typename` <*type*>" or `using`-directives likes "`using namespace` <*namespace-name*>".

Features UA extracts four features:

Type aliases These were very easy to extract since LibASTMatchers provides a pre-built matcher for this: `typeAliasDecl`. The only thing I had to improve was to ensure that the type alias was not part of a type alias template because I wanted to be able to report templated and nontemplated aliases separately.

Type alias templates These can be extracted using the `typeAliasTemplateDecl` matcher, which I also used above to filter the aliases from the alias templates. Recall that when you instantiate an alias template, you actually instantiate the class that is aliased and don't create a new type alias in the AST [28].

"`Typedef` templates" To distinguish "`typedef` templates" from class templates that happen to contain a `typedef` I made the following assumption about the idiom: A class template is considered a "`typedef` template" if and only if it contains

- at least one `typedef`,
- no declarations other than type definitions and access specifiers (`public`, `private` etc.).

I think that it is sensible to assume that a "`typedef` template" should only contain type definitions. If it contains anything else it is simply a regular class template. To account for the fact that according to the above such templated type definitions can contain multiple `typedef`s, a templated type definition is reported as often as the number of `typedef`s it contains.

```
template<typename T>
struct Wrapper {
    typedef std::vector<T> Tvector;
    typedef std::unique_ptr<T> Tpointer;
};
```

Right now, `Wrapper` will be reported twice because it contains two type definitions.

`Typedefs` Although type definitions are conceptually simple, they are the most difficult to extract correctly due to the risk of false positives: I don't want to report any type definitions here that are part of a "`typedef` template" because I want to clearly distinguish between type definitions and the type definition template idiom. When a "`typedef` template" is used, the compiler creates a specialization of its class template. This is a problem, because if we don't take care, the therein contained type definitions will be reported here. Just like an instantiation of an alias template doesn't create an alias that is matched by the tool, I don't want an instantiation of a "`typedef` template" to cause a type definition to be matched. Thus, to recap, there are two instances where type definitions occur which we don't want to report as type definitions, and thus have to filter out:

- a `typedef` in a class template that we think is a typedef template → reported as part of "`typedef` templates"
- a `typedef` in "`typedef` template" specialization/instantiation → never reported

This is what an extracted synonym looks like:

```
{
    "kind": "Alias",  // type definition or type alias?
    "location": 4,    // line number
    "templated": true // templated?
},
...
```

Using `kind` and `templated` we can represent the four features that are extracted.

Statistics This is again refreshingly simple: for each feature, we count how often it occurs and call it a day.

Limitations

Synonymized types I think that it might be also interesting to report the type that was synonymized.

"`Typedef` templates" containing multiple `typedefs` Instead of reporting a templated type definition multiple times, once for each contained type definition, it would be an improvement to report the type definitions it contains while stating that they were part of a "`typedef` template". I can imagine that programmers still like to use the idiom because it allows to "bundle" types neatly inside a single class, which we could research better if we made that change.

4.1.10 Variable Template Analysis (VTA)

Similar to UA, this analysis extracts all occurrences of variable templates and the idioms used to imitate it before it existed in the language.

Features

Variable templates I was surprised to find out that LibASTMatchers did not contain a proper matcher for variable templates. Luckily, a variable template is easily spotted in the Clang AST as a `VarTemplateDecl`. Using the information found on [9], I was able to write my own matcher for variable templates using a one-liner.

`static` data member of class template A class template is considered an instance of this idiom when it contains one or more `static` variable declarations, but not any other kind of declaration. When a class template contains multiple `static` variables, it is reported multiple times to account for that.

`constexpr` function template A function template is considered to be an instance of this idiom when the following conditions hold:

- The template is `constexpr`.
- Its return type is not `void`.
- It contains zero or more variable declaration, but not any other kind of declaration.
- It has a return statement, which returns one of the variables declared or some other expression.

These assumptions might be too restrictive, but there isn't much information about this idiom. A "`constexpr` function template returning the desired value" could be in principle any `constexpr` function template.

In the end, we report the kind (variable template or idiom) and the location of the feature.

Statistics For each of the three features, we count the number of occurrences to be able to find out how programmers like to template their variables.

Limitations Information about variable templates seems relatively sparse, and I haven't had many encounters with it yet. A blog post (see [43]) wrote that they are useful in type traits (apart from having variable-precision constants), suggesting that it might be interesting to dive into the C++ STL (Standard Template Library) to see how they are used there.
I found contradicting information about the pre-C++14 idioms (see [29] and [34]) w.r.t. the `constexpr`-ness of `static` data members of class templates. I think that [34] declares all idioms `constexpr` because it originally proposed `constexpr` variable templates, only in a later revision was decided to remove that requirement. The keyword `constexpr` is a must for variable templates for them to be useful in programming type traits, but I think that it can be omitted when variable templates (or either of its idioms for that matter) are used to declare variable-precision constants. A future improvement of VTA might thus drop the `constexpr` requirement for the function template idiom for the sake of consistency.

4.2 Testing methodology

I have written for each analysis a set of unit tests to test that the basic feature extraction of the analyses is correct. For each feature that an analysis extracts I have collected instances of it in different contexts to test that it is detected in those. More difficult to test is that the analyses don't match code that they should not.

LLVM Integrated Tester I used LLVM's integrated tester[5] (`lit`) to create the unit tests. `lit` allows writing compact test cases by packing test case input and test instructions into a single file. These files will then be put in a test directory and automatically discovered by `lit`. The `RUN` lines indicate what steps `lit` should perform.

```
1 // RUN: clang++ %s -emit-ast -o %t1.ast
2 // RUN: %cxx-langstat --analyses=tia -emit-features -in %t1.ast -out %t1.ast.
      json --
3 // RUN: diff %t1.ast.json %s.json
```

[5] https://www.llvm.org/docs/CommandGuide/lit.html

```
4  template<typename T>
5  class Sub {};
6
7  template<typename T>
8  class Super {
9      Sub<T> w;
10 };
11 template class Super<int>;
```

In the above example, we want to test that the explicit instantiation of `Super` and the implicit instantiation of `Sub` are reported, where the latter is caused by the former.

1. We use `clang++` to preparse the input file for us. `%s` stands for the name of the current file. This is optional, but useful for input files that `#include` headers because it avoids having to specify those using compile commands.
2. The tool is run with analyses enabled accordingly.
3. Comparison of the output with the reference output.

This way, I collected around 80 files containing basic tests to stress the extraction process.

This approach does not allow testing the classes and functions I declared and used in the framework, leaving them practically untested. Also, I did not use `lit` to test the correctness of the computation of statistics from features. However, the probability of finding errors should be smaller there since the functions performing those computations are relatively simple.

I also applied the analyses of `cxx-langstat` to its own source directory and compared the extracted features by hand to ensure that they weren't completely wrong.

4.3 Known issues

Lambdas By default, Clang's `functionDecl` matcher matches functions, member functions incl. constructors and destructors, but also lambdas. The problem with the latter is that they are currently matched three times due to their representation in the Clang AST, possibly skewing statistics. A quick fix would be to completely ignore lambdas by changing the matcher for function declarations to:

```
functionDecl(unless(hasParent(cxxRecordDecl(isLambda()))))
```

Even better would be if a lambda was just matched once, because it intuitively represents a single unnamed function. Analyses that inspect functions such as CCA, CEA, and FPA are susceptible to this.

`static` class data members `static` class data members often need the definition to be out-of-line:

```
struct C {
    static int data; // declaration
};
int Widget::data = 3; // definition
```

Such code causes problems for CEA, namely because the `constexpr`-ness of a variable is reported twice – once for the declaration, once for the definition. By the same token, TIA reports two instantiations if the `static` member is a variable template with an out-of-line definition.

Chapter 5

Results of the Analysis of the LLVM Core Libraries

To test the usability of the tool, we run `cxx-langstat` on the core of the LLVM project. The core of LLVM contains the source- and target-independent optimizer [13], the code generation suite to translate the LLVM internal representation (IR) to machine code [14] as well as the specification of the LLVM IR itself [13]. We analyzed 75 Gigabytes of `.ast` files, which were generated by Lukas Gygi using `CppBuild` [37]. The version of the LLVM libraries analyzed was 11.1.0. It took approximately 25 minutes to extract the features from the AST files and one minute to compute the statistics on an Intel Xeon X7550 clocked at 2GHz.

In the next section, I first want to detail some results that I gathered about C++ features, and in the section following that I will discuss what insights that we can gain from that data incl. the limitations of the data and potential improvements.

5.1 Results

Range-based `for` loops Anectodes of C++ books and articles suggest high `for` loop prevalence. The data showed comparable usage of `for`, range-`for` and `do-while` loops, but `while` loops saw little usage.

loop	occurrences
`for`	9493
range-`for`	13115
`do-while`	9426
`while`	2592

Function parameters and move semantics Of all functions that create machine code (i.e., excl. function templates, incl. full/explicit function template specializations and function template instantiations) 41744 had at least one by-value/copying paramater, 29825 had at least one non-`const` lvalue-ref parameter, 13944 had at least one `const` lvalue-ref parameter, and only 190 had at least one rvalue-ref parameter. In total, there were 72936 functions. In the case of function templates, the respective numbers were 575, 422, 170, and 0 found in 885 function templates. Additionally, 108 function templates adopted at least one forwarding reference parameter. In relative terms, the prevalence of the different kinds of parameters is comparable, as seen in figure 5.1.

Of all value parameters constructed in calls to functions that were not elided, 79787 were constructed using the copy constructor, 80501 using the move constructor.

`std::move` and `std::forward` were used 2821 and 1131 times, respectively. This represents all calls to those function templates, which can be in different contexts, e.g., in a call expression, a member initializer list, or a return statement.

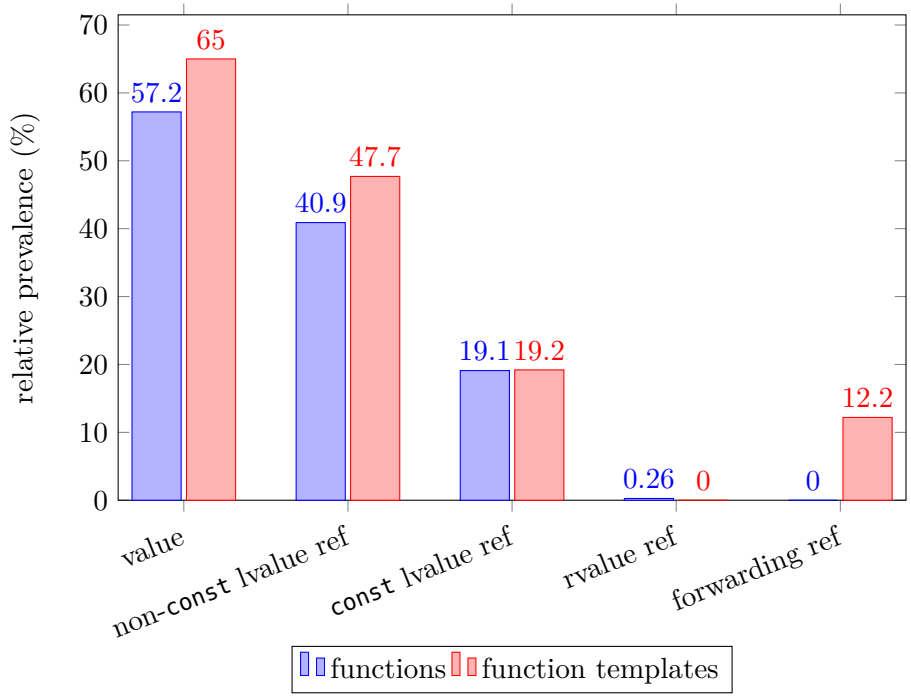


Figure 5.1: For each kind of parameter, % of functions/function templates that use it

Parameter packs When applying TPA to the LLVM core, we registered that a mere 1.1% of class templates, but a substantial 20.4% of function templates were using a parameter pack. Variable and alias templates came in both at 0%, which is due to the fact they virtually weren't used.

template	prevalence
class	1 out of 92
function	182 out of 893
variable	0 out of 1
alias	0 out of 0

`using` UA detected in total 1574 non-templated synonyms, of which 186 were type definitions and 1388 were type aliases, which is 11.8% and 88.2%, respectively. Interestingly, not a single templated synonym was found. The fact that no "`typedef` templates" were found could be due to the assumptions that I made about that idiom. I found it surprising to see that no alias templates were found, given that compared to type definitions one of their strong points is that they can be templated.

synonym	occurrences
`typedef`	186
alias	1388
"`typedef` template"	0
alias template	0

`constexpr` The following table shows for each C++ construct that can be adorned with `constexpr` the relative prevalence of that happening. Note again that this data does not establish a relation between how many constructs could be `constexpr` compared to how many actually were `constexpr` – it only filters out some trivial cases that are never constant expressions.

construct	occurrences
variable	327 out of 246628 (0.13%)
non-member function	6 out of 16824 (0.04%)
member function	83 out of 59837 (0.14%)
if statement	0 out of 147078 (0%)

Algorithms Of the total 1296 calls to algorithms found (see appendix for list of algorithms matched), essentially only the algorithms for getting the minimum and maximum are used.

Containers Figure 5.3 shows the results of applying CLA to the LLVM core. For each standard library container, it plots how many variable declarations and member variable

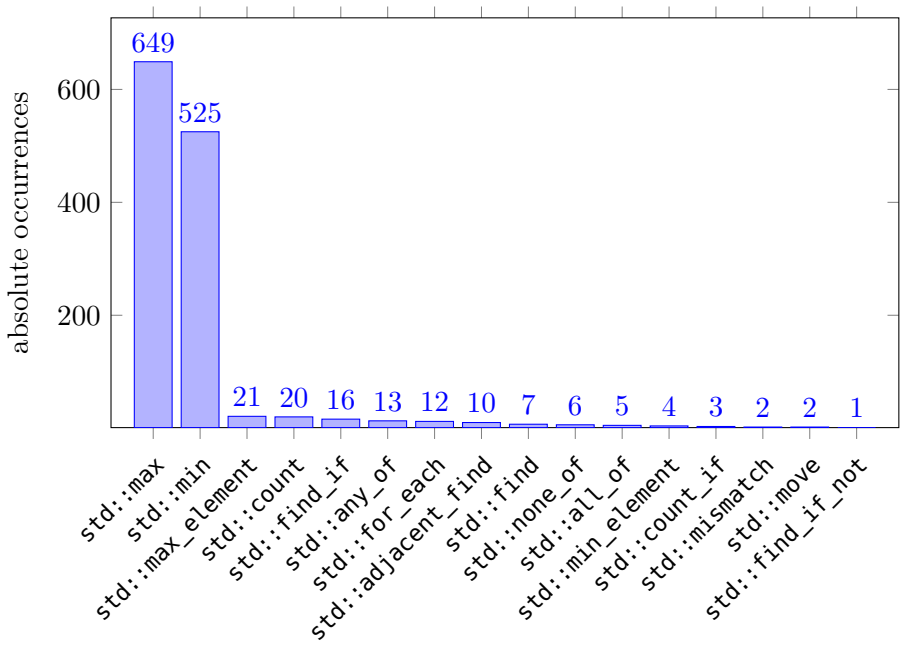


Figure 5.2: Number of occurrences of the analyzed algorithms

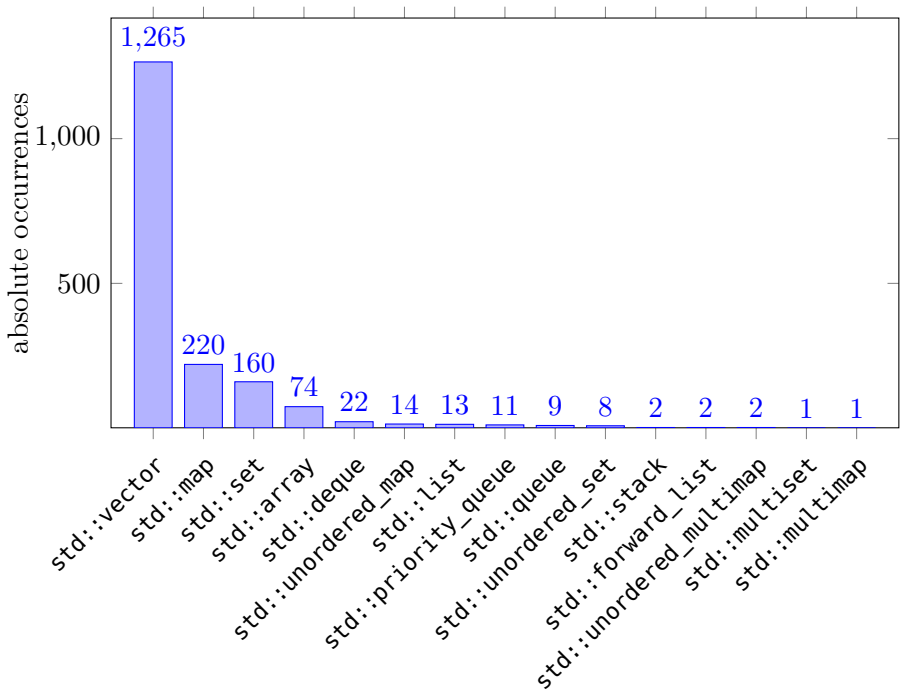


Figure 5.3: Number of occurrences of C++ Standard Library containers

declarations of that type were found in the code. This is of course only an approximation to the popularity or the container, but likely an important first step. According to this data, `std::vector`, `std::map` and `std::set` make up a whopping 70.2%, 12.2% and 8.9%, respectively of the 1802 containers used to declare variables, cumulatively responsible for more than 90%.
By looking at the instantiation arguments of `std::vector`, we get an idea of the contents of `std::vector` (see table 5.1). We ignore LLVM-specific types.

Although the instantiation arguments of the class templates are reported (allowing to get insights about container contents), I had to sift through those arguments by hand, which is too tedious.

contained type	occurrences
`std::vector`	25
`std::set`	5
built-ins	175
`std::unique_ptr`	32
`std::shared_ptr`	5
`std::weak_ptr`	1
`std::pair`	89
`std::tuple`	2

Table 5.1: `std::vector` contents of non-LLVM type

utility	occurrences
`std::unique_ptr`	1088
`std::shared_ptr`	162
`std::bitset`	34
`std::pair`	1311
`std::tuple`	54

Table 5.2: Number of occurrences of the analyzed utilities

This data probably doesn't paint a complete picture about the data structures used due to the LLVM-provided data structures [12] as well as the fact that containers can occur not only as variable declarations but also in return type declarations or literals which CLA (i.e., the underlying TIA) doesn't analyze.

Utilities and smart pointers There were 1250 variables/member variables declaring smart pointers in total, however, not a single `std::weak_ptr` variable. See table 5.2 for details.

LLVM containers LLVM provides its own set of containers, notably, a variety of `Small-` data structures that aim to reduce memory allocations on the heap [12]. Figure 5.4 shows the number of absolute occurrences of LLVM data structures declared as a (member) variable or parameter variable. These numbers come from an analysis built on top of TIA, so only templated class types are respected. The `Small-` class types (marked in blue in the plot) make up 61.4% of all occurrences.

5.2 Discussion

In this section, we discuss the results gathered and the ability of the framework to answer research questions and how to improve the results to do so better.

In the following, I present answers to research questions 1-3 and propose approaches to solving questions 5-7. I discuss an answer regarding question 4 but also propose an important improvement.

RQ1. Which new language features are easily and quickly adopted: range-based loops, move semantics, new keywords? Are there new standard components that are rarely seen in codes? Our data suggest high popularity of range-based loops, but I am not able to assess if there could be further increases in popularity. The data does not tell us if range-based loops were used at all locations where they could have been, making it harder to see how popular they effectively are. I would, however, argue that this is not as big of a problem as with `constexpr`, where there are many more conditions under which the construct is not applicable. Also, it would be beneficial to analyze multiple versions

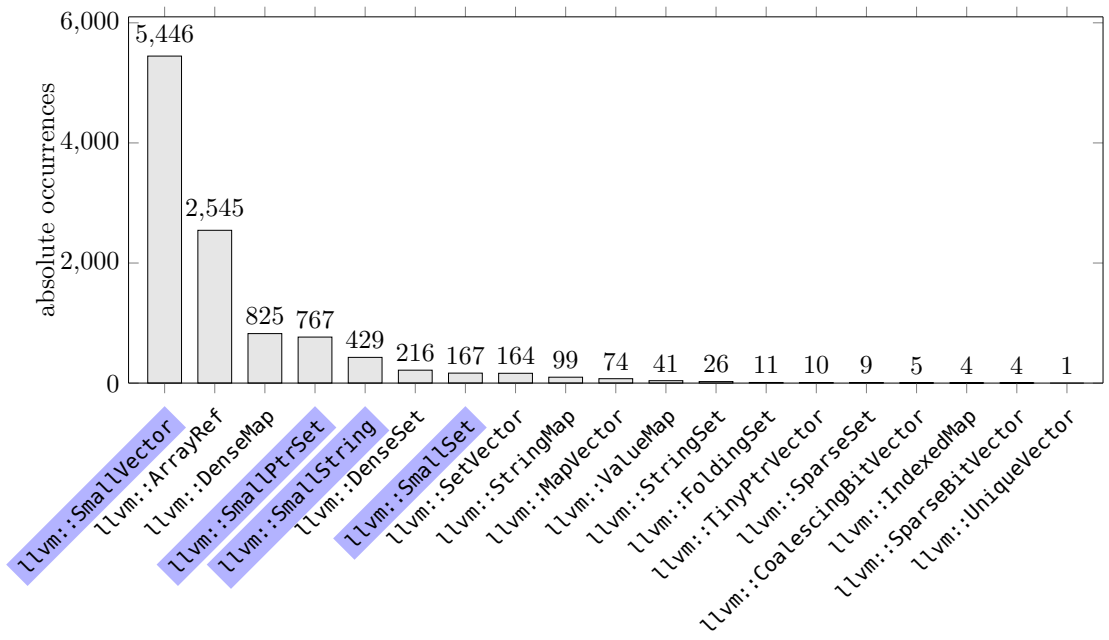


Figure 5.4: Number of occurrences of LLVM containers

of LLVM, ideally all versions from since the introduction of range-based loops until today and beyond to observe their adoption over time.

The adoption of move semantics at the call site to functions with by-value parameters is sizable at 50.2%. The existence of a move does not tell us why it occurred, thus we don't know whether we have to thank C++ and the compiler or the programmer for that efficient transfer of resources. The data does not disclose to us how many moves were performed because functions were called with an rvalue or whether the programmer decided to use `std::move`.
Even though we do have data on how often `std::move` is used, that data enumerates every use of it, which can be inside a call expression, but also a member initializer list or a return statement. This makes it impossible to figure out how many moves inside of call expressions were requested by the programmer.
The data about function parameters (around 60% of functions use by-value parameters) indicate that there is a potential gain to use move semantics. There is the problem that these by-value parameters also include those that cannot experience a performance boost using move semantics, e.g. language built-in types. This limitation, however, can partly overcome: the features contain information about the types of the parameters, which could be leveraged to filter out built-ins. This is an example where storing features and only then computing features has merit since it avoids expensive recomputations on the source code.

The adoption of parameter packs can be well analyzed by TPA, and the data suggests a sizable adoption of them among function templates. My guess is that template parameter packs are more popular with functions than with classes because one wants a function that accepts a variable amount of arguments more often than a class that stores variable amounts of data. To further investigate this assertion, we could improve FPA to also report on the commonness of function parameter packs.

I was surprised to see `constexpr` to be used so rarely – I would have expected it to be used more often adorning functions since requirements concerning function bodies have become less strict [41] (p. 100) since the first appearance of `constexpr` (member) functions, allowing more intuitive `constexpr` programming. These results should, however, be taken with a big grain of salt due to the limitations of CEA.

It is safe to say that type aliases are much more popular compared to type definitions, and it will be interesting to see if this trend continues if we analyze newer versions of this project and other projects in the future. I think that that will be the case, as the C++ community is shifting away from C-style programming.

RQ2. What is the usage rate of new standard modules and libraries, such as smart pointers and new containers? Using CLA and ULA, we analyzed the relative popularities of containers and container-like utilities provided by the C++ Standard Library. It shows that `std::vector` is hugely popular and that smart pointers also get their fair share. These analyses can also be used to investigate how many projects use the different types at all when analyzing multiple projects. However, an improvement would be to be able to relate their prevalence to that of non-container variables to see how many variables are containers in the first place. This would be beneficial to analyze smart pointer usage to see if they could catch up to raw pointers.
Also, there are many other places apart from (member) variables where these classes can be used, like inside other data structures, in return type declarations or literals. These occurrences are currently overlooked – table 5.2 suggests that `std::weak_ptr` isn't used, but there is one instance tucked away in an `std::vector`. This is supported by table 5.1, where we can see that we miss uses of containers and classes because the analysis looks only at the types of (member) variable declarations.

RQ3. Which algorithms are the most popular, which are not used at all? ALA solves this task well, and the results show us that apart from minimum/maximum operations, algorithms are used sparsely. A limitation that ALA looks at a smaller set of C++ Standard Library algorithms at the moment, but this isn't hard to resolve – the bigger problem is that it doesn't look at uses of function templates besides call expressions.

RQ4. Which non-standard libraries are most commonly spotted in projects? Can we deduce recommendations for new directions of standard development? Comparing the usage numbers of the LLVM `Small-` data structures with their C++ Standard Library counterparts (compare `llvm::SmallVector` to `std::vector` and `llvm::SmallSet`, `llvm::SmallPtrSet` to `std::set`) shows that select `Small-` containers are as popular if not more popular than the standard ones, hinting for such a development in the Standard Library. I would, however, argue that the data is a bit biased toward a high adoption of small data structures due to them being part of LLVM itself. It is necessary to analyze a larger set of C++ projects and consider a broader range of non-standard libraries that implement such containers (e.g. LLVM, Boost[1], etc.) to decide if the Standard Library needs such constructs.

For general, non-standard libraries, instrumenting TIA or applying it directly can give us usage statistics on any class and function templates, thus allowing us to detect the use of library components. The problem is that if a library relies more on nontemplated classes and functions rather than templated ones, TIA will not be able to analyze the library well. The problem lies in the name – TIA analyzes the usage of template instantiations only.
To remove this limitation, I propose to build a new analysis that contains two separate parts: one that makes use of TIA to analyze template usage and a second one that only analyzes the use of nontemplated class types and functions. The second part would operate similarly to TIA, the difference being that it explicitly looks for uses of nontemplated classes and functions. These two parts combined would make a flexible analysis that should be able to analyze all major components of a library.
For an example of a two-part analysis, see MSA (4.1.7).

[1] https://www.boost.org/doc/libs/1_75_0/doc/html/boost/container/small_vector.html

RQ5. How many projects involve any form of parallel processing? Which parallel frameworks are commonly spotted? `std::thread`, OpenMP, TBB, others? Currently, the framework has limited support to analyze the C++ Standard Threading Library, since it employs many non-generic classes and functions. A potential instrumentation of TIA couldn't analyze this library completely because it would only be able to analyze the usage of templates. TBB, on the other hand, can profit from an instrumentation of TIA, since many of its interfaces are programmed using templates [2].

RQ6. How prevalent are deprecated and unsafe constructs, best practices, and anti-patterns? We didn't gather much data concerning programming guidelines. I'm convinced that this framework is applicable to build analyses measuring such behavior. Take for example the recommended practice of using the `override` keyword to let the compiler (and colleague programmers, too) know that a virtual function is supposed to override the function of a base class. I see no deal-breaking problems implementing an analysis that first extracts all virtual functions, then relates base class and derived class functions by comparing their signatures and reports all instances where override is successfully used and all others where it isn't, but should be.

The `constexpr` analysis not only helps to identify the adoption of a C++11 feature but also checks the compliance with a programming practice – it seems Scott Meyers' advice to use `constexpr` relatively gratuitously [41] (p. 97) could be adhered to a little more.

RQ7. Which types of new features have the highest impact on new codebases and which are frequently adopted by projects that have been in development for a long time? Packing features into different categories such as keywords, library usage, language constructs and best practices we can see how applying this tool to multiple projects can give us insights into what kind of features are more gratuitously adopted by codebases and the programming community. Applying the tool to a single project across multiple versions would allow us to observe the prevalence of features as the project develops.
Of course, given I analyzed only a single version of one project, I can't give any concrete answers here.

Chapter 6

Conclusion

In this thesis, I have shown how I built a tool on top of Clang consisting of a framework and a set of analyses enabling to measure and analyze different types of C/C++ code features. We've seen how we extract features from source code, which we then use to compute statistics and why this is useful.

`cxx-langstat` already contains a sizable set of useful analyses, which I motivated with a broad background chapter about different features of C++. If necessary, new analyses can be added or even be built from others, e.g. in the same way that CLA is built by instrumenting TIA.

We've seen various results such as the overwhelming popularity of `std::vector`, the substantial adoption of range-based loops and type aliases as well as respectable use of parameter packs. Also, the results indicate that, as far as we can tell from the limited algorithms considered, how the C++ Algorithms Library suffers from low adoption.

By looking at a variety of research questions, I've demonstrated that `cxx-langstat` is flexible enough to encompass a variety of analyses that can help to answer those types of questions. For questions that couldn't be directly answered, I have argued how to extend existing analyses or create new ones.

Future work

For details about possible future research, see the discussion of research questions 4-7 in the previous chapter. This includes writing an analysis about class and function usage in general, including usage of both templated and nontemplated classes and functions as well as implementing an analysis to observe the compliance of programmers with best practice concerning the usage of the `override` and `final` keywords. Analyses about the usage of the `noexcept` keyword or the rule of three could also be added for more insights about the adoption of best practices.

Other opportunities for future work are found in the limitations paragraph contained in each analysis section, for example, the `constexpr` analysis from section 4.1.2, where work is needed to make the analysis only extract a location when `constexpr` is applicable.

Appendix A

Appendix

A.1 Full list of containers and algorithms analyzed

Containers

Sequence containers

- `std::array`
- `std::vector`
- `std::deque`
- `std::forward_list`
- `std::list`

(Unordered) associate containers

- `std::set`
- `std::map`
- `std::multiset`
- `std::multimap`
- `std::unordered_set`
- `std::unordered_map`
- `std::unordered_multiset`
- `std::unordered_multimap`

Container adaptors

- `std::stack`
- `std::queue`
- `std::priority_queue`

Algorithms

Non-modifying sequence operations

- `std::all_of`
- `std::any_of`
- `std::none_of`
- `std::for_each`
- `std::for_each_n`
- `std::count`
- `std::count_if`
- `std::mismatch`
- `std::find`
- `std::find_if`
- `std::find_if_not`
- `std::find_end`
- `std::find_first_of`
- `std::adjacent_find`
- `std::search`
- `std::search_n`

Minimum/maximum operations

- `std::max`
- `std::max_element`
- `std::min`
- `std::min_element`
- `std::minmax`
- `std::minmax_element`
- `std::clamp`

Modifying sequence operations

- `std::move` // not `std::move` from `utility`!

A.2 Flags for `cxx-langstat`

The source code can be found on GitHub: https://github.com/hartogss/cxx-langstat

Basic usage: `cxx-langstat [options]`. Append "`--`" and append zero or more compile commands on the command line or use `-p <build path>` to supply a compilation database in JSON format. See [15] for more information about compilation databases.

Options:

`--in`
Specifies an input file.
`--out`
Specifies an output file. Use only when a single output file is expected.
`--indir`
Specifies an input directory. Tool will run on all files in it, which ones depends on the stage.
`--outdir`
Specifies an output directory. Use when multiple output files are expected.

Stages:

`--emit-features`
Tool looks for source/AST inputs and will emit for each input an output containing extracted features.
`--emit-statistics`
Tool looks for JSON files (expects them to be generated by previous stage) and will emit a single output JSON file containing statistics.

`--analyses=<string>`
Comma-separated list of analyses to be run, can be supplied in either stage.
`--parallel` or `-j`
Accepts unsigned integer specifying how many batches to run parallel in the `--emit-features` stage. Ignored in the other stage.

Useful compile commands to be used after `--` :

`-ccc-install-dir`, `-resource-dir`
If you analyze source code (`.c`, `.cpp` or headers, not `.ast`) and you didn't install `cxx-langstat` in your `bin` folder, the `clang++` frontend might have trouble finding headers from the C/C++ Standard Libraries due to `clang++` thinking it is installed in the current build directory. Using `-ccc-install-dir /usr/local/bin` and `-resource-dir /usr/local/lib/clang/<version>` (paths might be different on your system) helped me resolve those problems. Also see[1].
You can also use `-I` to manually add include paths.

`--driver-mode`
Useful to force the frontend to interpret `.h` as C++ headers, however, deprecated. The `-x <language>` flag currently bugs, causing it to read no compile commands from the command line.

[1] https://gist.github.com/WingTillDie/f636de6e37947c2d23bf7400d6f24911

References

[1] Implicit instantiation (C++ only) — ibm.com. https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbclx01/implicit_instantiation.htm. [Online; accessed 21-March-2021].

[2] Intel® Threading Building Blocks Documentation. https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/introducing-the-intel-threading-building-blocks-intel-tbb.html. [Online; accessed 23-March-2021].

[3] Template instantiation (C++ only) — ibm.com. https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbclx01/template_instantiation.htm. [Online; accessed 21-March-2021].

[4] Working Draft, Standard for Programming Language C++, 2020.

[5] AST Matcher Reference — llvm.com. https://clang.llvm.org/docs/LibASTMatchersReference.html, 2021. [Online; accessed 21-March-2021].

[6] Clang 12 documentation, How to write RecursiveASTVistor based ASTFrontendActions — llvm.com. https://clang.llvm.org/docs/RAVFrontendAction.html, 2021. [Online; accessed 24-March-2021].

[7] Clang 12 documentation, Introduction to the Clang AST — llvm.com. https://clang.llvm.org/docs/IntroductionToTheClangAST.html, 2021. [Online; accessed 24-March-2021].

[8] Clang 12 documentation, LibTooling — llvm.com. https://clang.llvm.org/docs/LibTooling.html, 2021. [Online; accessed 21-March-2021].

[9] Clang 12 documentation, Matching the Clang AST — llvm.com. https://clang.llvm.org/docs/LibASTMatchers.html, 2021. [Online; accessed 21-March-2021].

[10] Clang 12 documentation, Tutorial for building tools using LibTooling and LibASTMatchers — llvm.com. https://clang.llvm.org/docs/LibASTMatchersTutorial.html, 2021. [Online; accessed 21-March-2021].

[11] Extra Clang Tools 13 documentation, Clang-Tidy — llvm.com. https://clang.llvm.org/extra/clang-tidy/, 2021. [Online; accessed 21-March-2021].

[12] LLVM Programmer's Manual — llvm.org. https://releases.llvm.org/11.0.0/docs/ProgrammersManual.html#picking-the-right-data-structure-for-a-task, 2021. [Online; accessed 17-March-2021].

[13] The LLVM Compiler Infrastructure — llvm.org. https://llvm.org, 2021. [Online; accessed 17-March-2021].

[14] The LLVM Target-Independent Code Generator — llvm.org. https://llvm.org/docs/CodeGenerator.html, 2021. [Online; accessed 17-March-2021].

[15] Eli Bendersky. Compilation databases for Clang-based tools. https://eli.thegreenplace.net/2014/05/21/compilation-databases-for-clang-based-tools, 2014. [Online; accessed 24-March-2021].

[16] Jonathan Bengtsson and Heidi Hokka. *Analysing Lambda Usage in the C++ Open Source Community*. Bachelor's thesis, Mid Sweden University, Department of Computer and System Science, 2020.

[17] Sy Brand. The Performance Benefits of Final Classes — microsoft.com. https://devblogs.microsoft.com/cppblog/the-performance-benefits-of-final-classes/, 2020. [Online; accessed 21-March-2021].

[18] Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. How c++ templates are used for generic programming: An empirical study on 50 open source systems. *ACM Trans. Softw. Eng. Methodol.*, 29(1), January 2020.

[19] cppreference contributors. Range-based for loop — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/range-for&oldid=125818, 2021. [Online; accessed 21-March-2021].

[20] cppreference.com contributors. Class template — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/class_template&oldid=121008, 2020. [Online; accessed 21-March-2021].

[21] cppreference.com contributors. Copy elision — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/copy_elision&oldid=125183, 2020. [Online; accessed 21-March-2021].

[22] cppreference.com contributors. if statement — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/if&oldid=123390, 2020. [Online; accessed 21-March-2021].

[23] cppreference.com contributors. Move constructors — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/move_constructor&oldid=125312, 2020. [Online; accessed 21-March-2021].

[24] cppreference.com contributors. Reference declaration — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/reference&oldid=118858, 2020. [Online; accessed 21-March-2021].

[25] cppreference.com contributors. std::forward — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/utility/forward&oldid=117989, 2020. [Online; accessed 23-March-2021].

[26] cppreference.com contributors. Template parameters and template arguments — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/parameter_pack&oldid=125154, 2020. [Online; accessed 21-March-2021].

[27] cppreference.com contributors. Templates — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/templates&oldid=123845, 2020. [Online; accessed 21-March-2021].

[28] cppreference.com contributors. Type alias, alias template — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/type_alias&oldid=125189, 2020. [Online; accessed 21-March-2021].

[29] cppreference.com contributors. Variable template — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/variable_template&oldid=125316, 2020. [Online; accessed 25-March-2021].

[30] cppreference.com contributors. constexpr specifier — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/constexpr&oldid=126029, 2021. [Online; accessed 21-March-2021].

[31] cppreference.com contributors. Containers library — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/container&oldid=126778, 2021. [Online; accessed 21-March-2021].

[32] cppreference.com contributors. The rule of three/five/zero — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/rule_of_three&oldid=126548, 2021. [Online; accessed 21-March-2021].

[33] cppreference.com contributors. Template parameters and template arguments — cppreference.com. https://en.cppreference.com/mwiki/index.php?title=cpp/language/template_parameters&oldid=127443, 2021. [Online; accessed 21-March-2021].

[34] Gabriel Dos Reis. Variable Templates (Revision 1), 2013.

[35] Gabriel Dos Reis and Bjarne Stroustrup. Templates Aliases, 2007.

[36] Peter Goldsborough. Example of McCabe (cyclomatic complexity) analysis. https://github.com/peter-can-talk/cppnow-2017/blob/master/code/mccabe/mccabe.cpp. [Online; accessed 24-March-2021].

[37] Lukas Gygi. *CppBuild: Large-Scale, Automatic Build System for Open Source C++ Repositories*. Bachelor's thesis, ETH Zurich, Zurich, 2021-01-28.

[38] Stephen Kelly. Exploring Clang Tooling Part 2: Examining the Clang AST with clang-query — microsoft.com. https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/, 2018. [Online; accessed 22-March-2020].

[39] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[40] Scott Meyers. Universal References in C++11. https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers, 2012. [Online; accessed 23-March-2021].

[41] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[42] Herb Sutter. Proposed Addition to C++: Typedef Templates, 2002.

[43] Baptiste Wicht. Simplify your type traits with C++14 variable templates. https://baptiste-wicht.com/posts/2017/08/simplify-your-type-traits-with-c%2B%2B14-variable-templates.html, 2017. [Online; accessed 25-March-2021].

[44] Steve Zdancewic and Zhendong Su. Lecture 10 Compiler Design. https://people.inf.ethz.ch/suz/teaching/252-0210-f20/lec10.pdf, 2020. [Online; accessed 28-February-2021].

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Code-driven Language Development: Framework for Analysis of C/C++ Open-Source Projects

Authored by (in block letters):

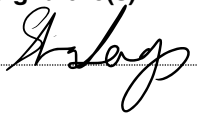
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hartogs	Siegfried

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 28.03.21	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.