FUNQUAL: USER-DEFINED, STATICALLY-CHECKED CALL GRAPH CONSTRAINTS IN C++

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Andrew Nelson

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Funqual: User-Defined, Statically-Checked Call Graph Constraints in C++
AUTHOR:	Andrew Nelson
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Professor of Computer Science

ABSTRACT

Funqual: User-Defined, Statically-Checked Call Graph Constraints in C++

Andrew Nelson

Static analysis tools can aid programmers by reporting potential programming mistakes prior to the execution of a program. Funqual is a static analysis tool that reads C++17 code "in the wild" and checks that the function call graph follows a set of rules which can be defined by the user. This sort of analysis can help the programmer to avoid errors such as accidentally calling blocking functions in time-sensitive contexts or accidentally allocating memory in heap-sensitive environments. To accomplish this, we create a type system whereby functions can be given user-defined type qualifiers and where users can define their own restrictions on the call graph based on these type qualifiers. We demonstrate that this tool, when used with handcrafted rules, can catch certain types of errors which commonly occur in the wild. We claim that this tool can be used in a production setting to catch certain kinds of errors in code before that code is even run.

ACKNOWLEDGMENTS

Thanks to:

- Dennis Ritchie and Bjarne Stroustrup. You've accidentally created something hauntingly expressive, painstakingly verbose, ingeniously strict, and idiotically sloppy. C++17 is a hot mess but it's everywhere — thank God it's type safe.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Writing bug-free software is challenging if not impossible. In the past 30 years, millions of dollars have been invested in tools that help developers write code that is robust, readable, and correct [18]. In general these tools fall into two categories: dynamic analysis tools such as gdb, valgrind, and IDA which analyze programs as they are running; and static analysis tools such as lint, cppcheck, and `gcc -Wall` which analyze programs before they are run. All these tools have different use cases and can be used together to minimize the presence of errors in code.

While these tools are extremely helpful in finding bugs in code, they are by no means complete. Every tool uses a finite set of techniques to detect a specific class of issues. Some tools examine the types of values and expressions to enforce type safety[18], some tools examine ownership of objects to enforce memory safety[11], some tools examine the flow of values through a program to ensure security[8], and many other tools do other things entirely.

This paper intends to add a new technique to the existing arsenal. This tool makes it possible to check for errors which were previously undetectable. To motivate this technique, we provide a problematic example. Figure 1.1 contains a snippet of C code that has a bug in it — the reader is challenged to find it.

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo) {
    printf("Received signal %d\n", signo);
}

int main(void) {
    if (signal(SIGINT, sig_handler) == SIG_ERR) {
        printf("Could not register signal handler\n");
        return 1;
    }

    printf("Signal handler registered...\n");
    while (1) {
        printf("Waiting for signals...\n");
        sleep(1);
    }
}
```

Figure 1.1: Example piece of C code containing an error

Most well-seasoned C and C++ programmers would be at a loss to find the error — and the error certainly is obscure. A quotation from the glibc library reference may be helpful here:

> If a function uses a static variable or a global variable, or a dynamically-allocated object that it finds for itself, then it is non-reentrant and any two calls to the function can interfere [10].

By "two calls", the reference means two concurrent calls. In the above snippet of code, a `SIGINT` signal sent to the process preempts whatever function was currently executing and transfers execution to `sig_handler`. `Sig_handler` proceeds to call `printf` which may or may not already be executing in the main context. This is problematic because `printf` grabs a global lock around `stdout` and in the case of concurrent calls results in deadlock. Not good.

The glibc library reference goes on to explicitly mention several common functions as being nonreentrant. A few of them are `malloc`, `free`, `fprintf`, `printf`, and any function that modifies the global `errno`, although any function which uses static, global, or dynamically-allocated state will fall into this category.

A stop-gap measure that could be implemented to solve this issue is to make a rule: *No interrupt handlers are allowed to call nonreentrant functions*, and to ask your peers to inspect all code by hand to enforce this requirement. This is tedious, error-prone, and can be extremely difficult for code at scale. Let's say, for instance, that `sig_handler` called `foo`, and `foo` called `bar`, and `bar` called `printf`. Is it reasonable to expect a human to detect this error in judgment that occurred through 4 layers of indirection? Probably not.

To solve this problem, and many others like it, we created a tool called funqual. Funqual allows C++ programmers to tag certain functions and will statically check

the call graph and function tags against a set of user-defined rules. This call graph type system is totally orthogonal to the existing C++ type system and so does not interfere with or expand the existing type rules which should be familiar to C++ programmers. Instead, funqual provides an additional set of restrictions which, when used intelligently by the developer, can help to detect certain kinds of errors statically.

Funqual is written using libClang and does not require any additions to the syntax of C++. As such, funqual can be run on C++17 code "in the wild" (code not designed to work with funqual); additionally, code which has been annotated for use with funqual can be compiled directly with gcc or clang without any modification.

This thesis is laid out as follows: Chapter 2 covers background information and formally develops the concepts of a call graph and an indirect call. Chapter 3 covers related work in such a way as to contrast the techniques of funqual from the techniques used by other tools in this domain. Chapter 4 gets into the theoretical details of how the type system in funqual works including a high level overview, an in-depth explanation of each individual rule, and some formal arguments for correctness. Chapter 5 goes into the practical details about the implementation and usage of funqual. Chapter 6 demonstrates funqual in action by showing how to apply it in some real-world projects. Finally, Chapter 7 discusses future improvements that can be made to funqual and Chapter 8 offers a conclusion.

Chapter 2

BACKGROUND

This Chapter aims to provide context for funqual as well as to provide an intuition for why funqual works the way that it does. Section 2.1 presents a brief review of type systems that should be familiar to most programmers; special care is taken to define systems of type qualifiers. Section 2.2 develops the concept of a call graph and sets the stage for the two concepts to be combined later in the paper.

2.1. Classic Type Qualifiers

In most research into type-systems, type qualifiers are a way to refine variable types in order to introduce additional constraints. These type qualifiers can generally be applied to any base type and can often be combined to form even more specific types. A classic example that most programmers of C-family languages will know is the `const` type qualifier. Any identifier with the `const` qualifier can be initialized with a value but can never be assigned to again. This restriction can be statically checked and can often help prevent certain types of errors when used intelligently by the programmer [5]. Another type qualifier which may be familiar to C programmers is `volatile` which tells the compiler (and programmer) that this variable may be changed suddenly by other execution environments [5]. The important thing to note is that the rules surrounding these type qualifiers are orthogonal to the rules of the main type system. A `const` identifier is treated the same way whether it a `const int` or a `const char*` or a `const Panda` or even a `const volatile int` — the *type* and the *type qualifiers* exist in separate type systems and so the rules are enforced separately.

Some compilers also have their own compiler-specific type qualifiers. In Microsoft Visual C++, function parameters that are modified by the caller and referenced by the callee can be annotated with the `[Runtime::InteropServices::Out]` qualifier to tell the programmer and the compiler that this is an out parameter. Having a programming environment rich in these type qualifiers can help make the intent of source code easier for the programmer to infer and make it possible for those intents to be statically checked by the compiler.

In the majority of these systems, defining additional type qualifiers is either relegated to the language designers or to the compiler maintainers. There is not much tooling or support for the average programmer to create their own type qualifiers and there does not seem to be any sort of emphasis on creating project-specific qualifiers to help maintain program semantics.

2.2. Turning Program Source into a Call Graph

The focus of this paper is on creating and analyzing type qualifiers for functions that constrain where those functions can and cannot be called. The central notion behind this sort of type checking is that every program has a call graph and that there are certain patterns in the call graph which must be prevented.

A program's call graph is a directed graph where each function is a vertex and where each call is an edge directed from the caller to the callee. The type qualifiers in this context are applied to the vertices and the things we wish to constrain are connections between vertices. Below is an example of a C program and its associated call graph.

As demonstrated in Figure 2.2, every function in the source code has a vertex in the graph and every function call in the source code has an edge in the graph. If there is a call from function X to function Y in the source code, there will be an edge

```c
int breed_and_release_pandas() {
    Panda *baby_panda = malloc(sizeof(Panda));
    release_panda(baby_panda);
}

int save_the_pandas() {
    stop_deforestation());
    if (pandas_are_saved()) {
        printf("Stopping deforestation saved the pandas!\n");
        return 1;
    }

    breed_and_release_pandas();
    if (pandas_are_saved()) {
        printf("Breeding pandas in captivation and releasing them has saved the pandas!\n");
        return 1;
    }
    return 0;
}

int main(void) {
    if (save_the_pandas()) {
        printf("The pandas have been saved!\n");
    }
}
```

Figure 2.1: Example C program. The call graph for this program is shown in Figure 2.2

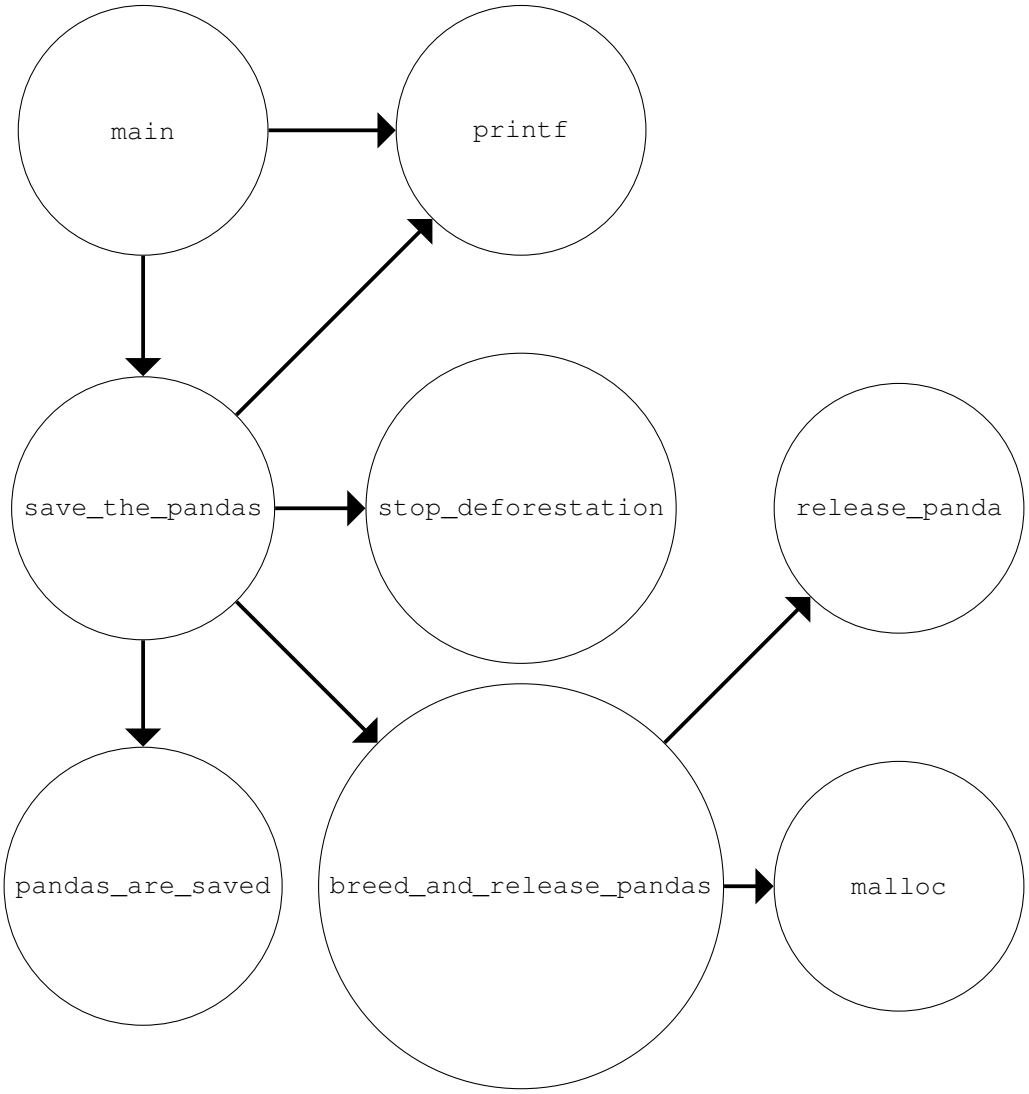

Figure 2.2: Example Call Graph. The source code associated with this call graph is shown in Figure 2.1

pointing from node X to node Y in the associated call graph.

This graph representation makes it easy to reason about the program algorithmically. Does `main` contain a call to `breed_and_release_pandas`? No. You can tell because there is no edge from `main` to `breed_and_release_pandas`. Does `breed_and_release_pandas` contain a call to `release_panda`? Yes. You can tell because there is an edge from `breed_and_release_pandas` to `release_panda`. Does `save_the_pandas` indirectly call `malloc`? Yes. You can tell because there is a path from `save_the_pandas` to `malloc`. Thanks to the call graph, questions about which functions call which functions boil down to classic path finding algorithms.

Chapter 3

RELATED WORK

Static program analysis is a hot topic in Computer Science research. The Association for Computing Machinery (ACM) publishes several journals, such as Proceedings of the ACM on Programming Languages (PACMPL), Transactions on Programming Languages and Systems (TOPLAS), and Transactions On Software Engineering and Methodology (TOSEM), which are focused (at least in part) on static verification and type systems. It should come as no surprise that there is a large body of research that is related to this thesis. This Chapter references a tiny fraction of this body of work. Section 3.1 calls upon past research to assert unquestionably the positive impact that static analysis has on the software development process. Section 3.2 explores a line of research dedicated to inserting supplemental specifications into existing programming languages in order to improve the static checkability of those languages. Lastly, Section 3.3 pays respect to the LLVM project which has enabled so much of the research for this thesis.

3.1. On the Effectiveness of Static Analysis

Studies have long shown that static analysis is an essential tool for developing high-quality software. The high speed and low cost of this type of verification make it an economical method for finding faults in program code [18, 12].

Industry has taken this observation to heart. Many companies have their own internal tools dedicated to statically checking code changes with a goal of detecting common mistakes and stylistic issues. The Mozilla project is a good example of this — since the early 2000s, Mozilla has used a fairly robust suite of internal tools

specifically crafted for Mozilla's mostly C++ codebase. Using these tools, every Pull Request into Mozilla Firefox is parsed and checked against a set of hand-written rules to detect and report common issues [6, 1]. Much of this tooling was dedicated to detecting memory issues. Of course, without modifying the grammar of C++, there are limitations in what can be easily checked statically by these tools. Only a small subset of the problem could be effectively detected.

More recently, Mozilla developed and began using a language called Rust which was designed with certain static analysis characteristics in mind. The Rust language implements an innovative type system meant to formally track the ownership of objects in memory. "Rust's type system and runtime guarantee the absence of data races, buffer overflows, stack overflows, and access to uninitialized or deallocated memory" [11]. A common sentiment in the Rust-language community is that even though the "Borrow Checker" (the part of the type system that enforces memory safety) seems complicated at first, seasoned Rust users learn to depend on it to help them reason through complicated programs [17]. Rust demonstrates that making a type system more expressive and more restrictive can improve both the static checkability of a programming language and also the help the users of those languages.

3.2. Aftermarket Type Systems — Supplementing an Existing Language

The idea of introducing new forms of type checking into an existing language to increase safety is nothing new. As early as 1994 tools such as LCLint have existed which allow the programmer to write down specifications about their code that are not necessarily supported by the original language standard. The LCLint tool can take program source code as well as a file containing supplemental specifications and perform static analysis that is more thorough and informed than could possibly be achieved based on the language standard alone [4].

A useful attribute of these supplemental static analysis tools is that they scale incrementally — the programmer can use these tools to whatever extent they find helpful and can increase or reduce the amount of information they pass on to these tools as they see fit. Since these specifications are opt-in, adding new forms of specification to a tool like LCLint is a straightforward way to expand the scope of the tool without breaking backwards compatibility. As an example, in 1996, Evans *et al.* added a few variable type annotations to LCLint such as `not-null`, `possibly-null`, and `null`. When used by the programmer, these annotations allow LCLint to check for certain kinds of errors relating to nullness and memory allocation [3]. Such modifications require zero action by the users that choose not to use them; if a variable is not annotated then LCLint will not try to check that variable. However, as the user adds more annotations, LCLint is able to check more variables. The amount of feedback LCLint is able to provide scales up and down with the amount of annotations in the code.

In general, variable annotations like `not-null` and `possibly-null` are very similar in use to the existing system of type qualifiers in the C family of languages. A canonical example of a type qualifier would be the C `const` qualifier; a variable marked `const` may be set once at declaration but never updated again (ignoring unsafe casts). Type qualifiers and annotations like `const` and `not-null` have two benefits: First, they declare the intent behind the code so that other programmers reading the code have a better idea of how it works. Second, they dictate what the programmer can and cannot do with an identifier so that the compiler or other static checking tool can detect accidental misuse. However, their use is entirely optional — the programmer can choose to treat an identifier as `const` or `not-null` without actually adding the annotation [5].

"A Theory of Type Qualifiers" develops this concept in depth and explores the theoretical and practical concerns involved with using type qualifiers in a language [5].

One of the most relevant observations to the work in this paper is that every type qualifier introduces a form of subtyping. For all types T and any qualifier q, either $T \leq qT$ or $qT \leq T$ depending on q[1]. Here we notate T qualified by q as qT and we notate X is a subtype of Y as $X \leq Y$. $X \leq Y$ should be interpreted to mean that X can be safely used whenever Y is expected. For example `int` $\leq$ `const int` because in any statement containing a `const int`, one could safely substitute an *int* however the reverse is not true. In the same vein, `not-null char*` $\leq$ `char*` because any statement referencing a `char*` could safely be given a `not-null char*` instead [5]. In this paper we will apply this concept to the type qualifiers introduced by funqual in order to argue for the correctness of funqual.

3.3. libClang and the Explosion of C++ Tooling

C++ is difficult to parse [9, 15, 13, 14]. Years of language additions, the need for backwards compatibility, and the existence of a text-based preprocessor[2] means that the language grammar is large and complicated. As a result, even the simplest static analysis tools require a huge amount of complexity to do basic parsing of source code. Up until relatively recently, many C++ language tools settled on doing a partial parse of the language using approximations and heuristics [15]. This method can lead to artificial constraints on the language or to incorrect interpretations of the source.

As a result of the LLVM Compiler Infrastructure Project, we now have an excellent set of tools for working with code. The Clang compiler is a fully featured compiler from the LLVM project that supports a wealth of C-family language standards including C++17. The LLVM project also provides libClang which exposes a convenient

[1]This notation is equivalent to the subset notation (i.e., $T \subseteq qT$ or $qT \subseteq T$). We choose to use the less than operator because it matches the notation used by Foster in "A Theory of Type Qualifiers".

[2]In theory, preprocessing could be delegated to another tool like gcc. In practice this generally leads to loss of information — most notably with `#include` directives obfuscating the locations of symbols in source code.

API to the parser and the AST used by the Clang compiler. libClang enables developers to create their own tools that build on top of Clang's C++ parser. This means that developers of static analysis tools only need to focus on maintaining their project's contributions rather than supporting an entire parser/AST toolsuite [15]. Funqual is built using libClang and so the work done in this paper was only possible thanks to the work done by the LLVM Compiler Infrastructure Project.

Chapter 4

TYPE RULES

Funqual checks a program against a type system. It is a tool that takes in source code as well some user-defined call graph rules, does some computation, and prints one of two things: "This program is well-typed", or "This program is not well typed" (in practice the later case also comes with an explanation as to why the program is not well-typed). If a program is well-typed, then it is free from call graph rule violations. If a program is not well-typed, then it may contain one or more errors.

This chapter contains an overview of the rules implemented by funqual as well as a brief exploration of what needs to happen behind the scenes in order to correctly check these rules. Section 4.1 demonstrates the big picture of what these rules are trying to accomplish. Section 4.2 explains how type qualifiers are applied to function pointers and how funqual checks them. Section 4.3 is a detailed explanation of each of the call graph constraints supported by funqual. Finally, Section 4.4 explains a few special cases and explains how funqual handles them to create a complete call graph.

Note that this chapter focuses only on the conceptual design of funqual and its type system. For details on how it is implemented or how to use it, refer to Chapter 5.

4.1. Overview

Before doing a deep dive into the specific rules of funqual, let us look at an example. Recall the save the pandas example from Section 2. It is reproduced in this Section as Figure 4.1 for convenience.

Let us now imagine that there is some constraint whereby `save_the_pandas` should not allocate memory. As programmers we would like to believe that we are

```c
int breed_and_release_pandas() {
    Panda *baby_panda = malloc(sizeof(Panda));
    release_panda(baby_panda);
}

int save_the_pandas() {
    stop_deforestation());
    if (pandas_are_saved()) {
        printf("Stopping deforestation saved the pandas!\n");
        return 1;
    }

    breed_and_release_pandas();
    if (pandas_are_saved()) {
        printf("Breeding pandas in captivation and releasing them has saved the pandas!\n");
        return 1;
    }
    return 0;
}

int main(void) {
    if (save_the_pandas()) {
        printf("The pandas have been saved!\n");
    }
}
```

Figure 4.1: Example C program. Running this code in a production environment may not actually save the pandas

disciplined enough to remember this rule and enforce it ourselves. In practice, self-regulation like this often ends poorly. As a result we would like a tool like funqual to enforce this constraint automatically.

Funqual allows us as programmers to create our own type qualifiers and to apply whatever meaning we want to those qualifiers. In this particular case we create two type qualifiers: `static_memory` and `dynamic_memory`. We also create one rule: $restrict\_indirect\_call(static\_memory, dynamic\_memory)$. When the programmer qualifies a function with `static_memory`, that declares the intent that this function will *never* allocate memory on the heap. When the programmer qualifies a function with `dynamic_memory`, that declares the intent that this function always allocates memory on the heap[1]. The rule $restrict\_indirect\_call(static\_memory, dynamic\_memory)$ tells funqual that `static_memory` functions are not allowed to call `dynamic_memory` functions either directly or indirectly. If it is possible for a `static_memory` function to reach a `dynamic_memory` function, then the rule has been violated and funqual should inform the user.

In the example about saving the pandas, we would qualify `save_the_pandas` as `static_memory` and we would qualify `malloc` as `dynamic_memory`. Figure 4.2 shows the call graph for Figure 4.1 with `static_memory` functions marked green and with `dynamic_memory` functions marked red.

By turning the program into a directed graph and by assigning types to the vertices, we have transformed the problem of type qualifier rule satisfaction into a graph problem. A question like *are there any static_memory functions that inadvertently call dynamic_memory functions* essentially boils down to *are there any paths from green vertices to red vertices*. In this example, the answer to that question

[1]The meanings of these type qualifiers are as determined by the programmer; without a rule to operate on them, funqual will completely ignore the type qualifiers.

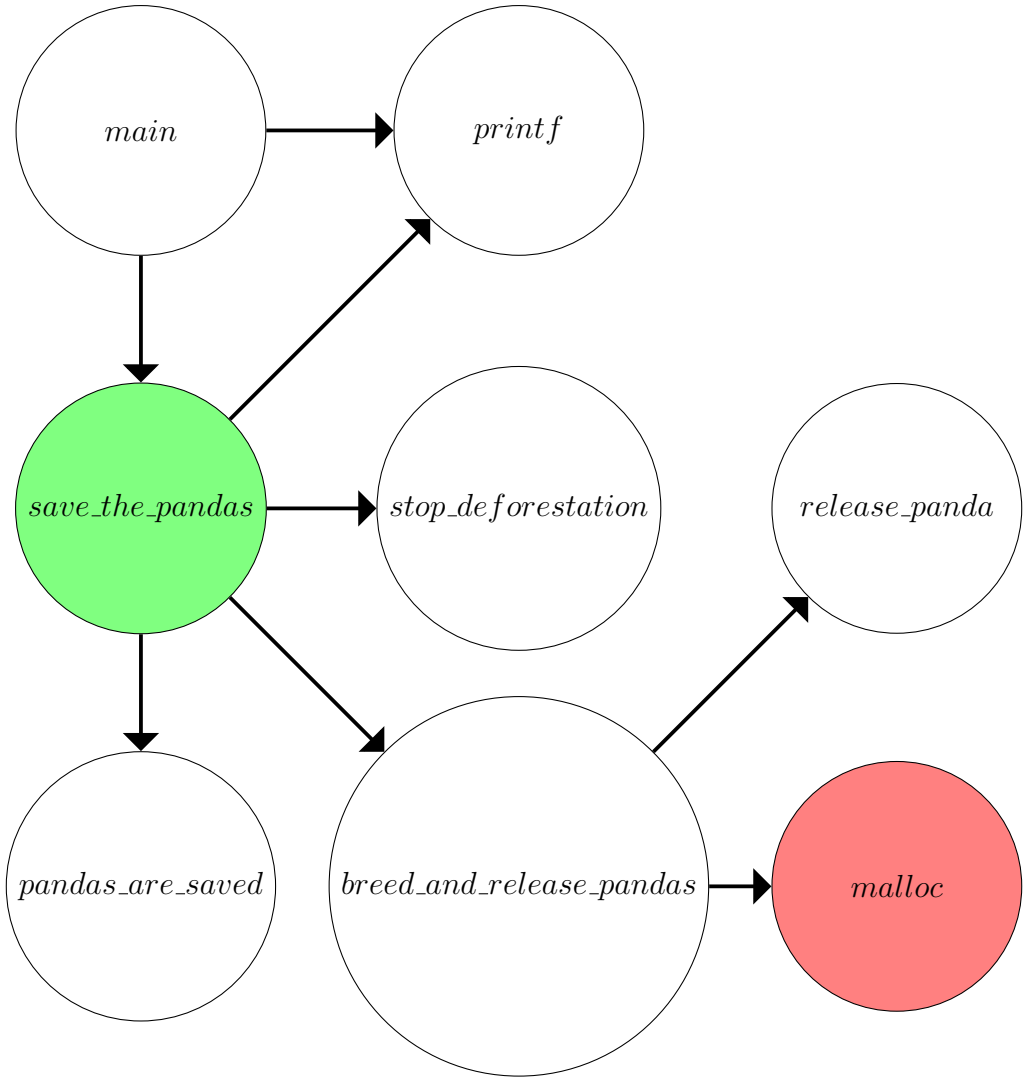

Figure 4.2: Color-coded Call Graph for Figure 2.1. Functions tagged `static_memory` are highlighted green and functions tagged `dynamic_memory` are highlighted red.

is yes. In the code, `save_the_pandas` calls `breed_and_release_pandas` which calls `malloc` constituting an illicit call. Equivalently, `save_the_pandas` has an edge to `breed_and_release_pandas` which has an edge to `malloc` constituting an illicit path. A well-typed program has no paths from green vertices to red vertices. A poorly-typed program will have at least one path.

4.2. Function Pointers and Indirect Type

Traversing a program for function calls and adding them to the call graph is relatively straightforward. Knowing exactly what function is being called at the time of parsing makes this process trivial. This does not account for all function calls, however. There are multiple cases in modern C++ where a function call is either happening behind the scenes or where the exact callee is not knowable. This section examines function pointers and explains how they are represented in the call graph.

As a concrete example, refer to Figure 4.3. In this example, it is literally impossible to know which function `strat` is going to point to. This is a pointed example, but `rand` can represent any expression whose result is unknowable during static analysis. Additionally, in this example there are very clearly only three functions that `strat` could point to. In a real program, there might be thousands of functions and they might not all be listed in one place.

If we still intend to use funqual to enforce this *restrict_indirect_call*(*static_memory*, *dynamic_memory*) rule then we are going to need some additional tools. Since keeping track of all the possible values of `strat` is impractical, we will instead keep track of the type of `strat` with respect to this call graph. Recall that the type of `save_the_pandas` is `static_memory` and that the type of `malloc` is `dynamic_memory`. If we had a function pointer pointing to `malloc`, then the type of that function pointer would have to also be `dynamic_memory`. In this example we

```c
int breed_and_release_pandas() {
    Panda *baby_panda = malloc(sizeof(Panda));
    return release_panda(baby_panda);
}

int (*)() get_random_strategy() {
    switch (rand() % 3) {
        case 0:
            return breed_and_release_pandas;
            break;
        case 1:
            return stop_deforestation;
            break;
        case 2:
            return stop_hunting;
            break;
    }
}

int save_the_pandas() {
    while (!pandas_are_saved()) {
        int (*strat)() = get_random_strategy()
        strat();
    }
    return 0;
}

int main(void) {
    return save_the_pandas();
}
```

Figure 4.3: In this example C program, it is impossible to know statically what the value of `strat` is. Because of this, funqual requires the programmer to annotate function pointers with additional type information.

have a function pointer pointing to `breed_and_release_pandas`. We will say that `breed_and_release_pandas` has *indirect type* `dynamic_memory` because it calls malloc and so any function pointer that points to `breed_and_release_pandas` must have indirect type `dynamic_memory`.

For this reason, when we use function pointers we will have two kinds of type qualifiers: *direct type* qualifiers and *indirect type* qualifiers. Direct type refers to the funqual type qualifiers we have explicitly assigned to the pointee. Indirect type refers to the funqual type qualifiers of all the functions reachable from the pointee. Direct type for both functions and function pointers must be explicitly annotated in the code. Indirect types for function pointers must be annotated explicitly but indirect types for functions can be inferred.

Figure 4.4 shows the same code as Figure 4.3 but with function types annotated. Figure 4.5 shows the call graph for Figure 4.4 with the function pointer represented as a cloud. Notice that we do not need to write any explicit annotations for the indirect type of `breed_and_release_pandas`. Funqual has all the information it needs to statically infer the indirect type of functions. In this case, the indirect type is `dynamic_memory` because `breed_and_release_pandas` calls `malloc`. Also notice that `strat` has indirect type `dynamic_memory`. This matters because it is *possible* that calling `strat` might result in a `dynamic_memory` function getting called.

Thanks to the graph based representation of this program, it is clear to see where the error is. `save_the_pandas` calls `strat` and it is possible that a call to `strat` could result in a call to `malloc`. The indirect type of `strat` (notated in Figure 4.5 as red horizontal lines) is how we keep track of this possibility.

```
int breed_and_release_pandas() {
    Panda *baby_panda = malloc(sizeof(Panda));
    return release_panda(baby_panda);
}

int indirect_dynamic_memory (*)() get_random_strategy() {
    switch (rand() % 3) {
        case 0:
            return breed_and_release_pandas;
            break;
        case 1:
            return stop_deforestation;
            break;
        case 2:
            return stop_hunting;
            break;
    }
}

int save_the_pandas() static_memory {
    while (!pandas_are_saved()) {
        int indirect_dynamic_memory (*strat)() =
            get_random_strategy()
        strat();
    }
    return 0;
}

int main(void) {
    return save_the_pandas();
}
```

Figure 4.4: Same example program as Figure 4.3 but with function pointer type annotations inserted

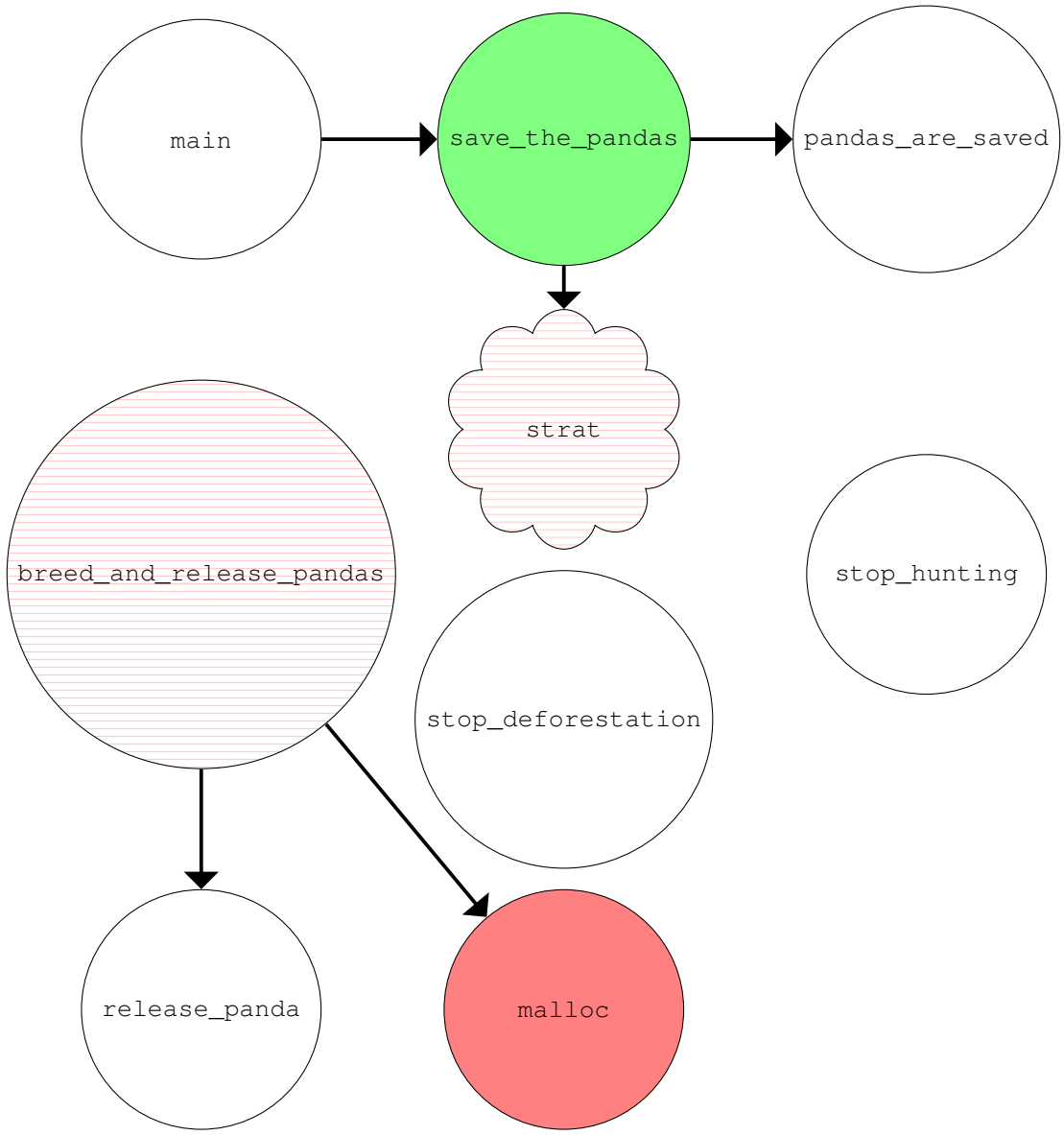

Figure 4.5: Color-coded Call Graph for Figure 4.4. Functions tagged `static_memory` are highlighted green and functions tagged `dynamic_memory` are highlighted red. Indirect types are represented as horizontal line patterns on a node. Clouds represent function pointers.

4.2.1. Rules of Assignment

To properly enforce call graph constraints, funqual checks function pointers in two places: first when the function pointer is assigned, and second when the function pointer is called. The rules described in this section are crafted specifically to maintain call graph correctness. For the purpose of this discussion, we will let L stand for some function pointer and we will let R stand for some function value (the names L and R are a reference to the `lvalue` and `rvalue` in a typical assignment statement).

When assigning a function pointer L to point to a function R, there are two rules that funqual checks: the direct type of L must match exactly the direct type of R, and the indirect type of L must be a superset of the indirect type of R. For function pointers, both the direct and indirect types must be explicitly annotated in code. For functions, only the direct type must be explicitly annotated as the indirect type can be inferred.

These rules are necessary to maintain the soundness of the system. In order to correctly enforce $require\_direct\_call(X, Y)$, the direct type of L must be contained in R — otherwise a call to L might be considered valid even if R does not have Y in its type. In order to correctly enforce $restrict\_direct\_call(X, Y)$, the direct type of R must be contained in L — otherwise a call to L might be considered valid even if R does not have Y in its type. Combining both of these requirements means that the direct types of L and R must match exactly. Lastly, in order to properly enforce $restrict\_indirect\_call(X, Y)$, we need to know all the funqual types that are possibly reachable by calling L.

Table 4.1 shows a few examples of valid and invalid assignments.

lvalue direct	lvalue indirect	rvalue direct	rvalue indirect	Valid?
(none)	(none)	(none)	(none)	Valid
static_memory	(none)	(none)	(none)	Not Valid
(none)	(none)	static_memory	(none)	Not Valid
static_memory	(none)	static_memory	(none)	Valid
static_memory	blocking	static_memory	(none)	Valid
static_memory	(none)	static_memory	blocking	Not Valid
static_memory	blocking	static_memory	blocking	Valid
static_memory	blocking	static_memory	nonblocking	Not Valid
static_memory	blocking nonblocking	static_memory	nonblocking	Valid
(none)	blocking static_memory nonblocking	(none)	(none)	Valid

Table 4.1: Examples of valid and invalid assignments in funqual. The left two columns show the direct and indirect type of the lvalue respectively. The next two columns show the direct and indirect type of the rvalue respectively. The rightmost column shows whether or not that assignment is valid.

4.3. Call Graph Rules

Each subsection here describes one of the call graph rules supported by funqual. For each rule, we explain the meaning, provide an algorithm that could enforce it, and present an argument for the algorithm's correctness with respect to the rest of the type system. The algorithms presented here only return `true` or `false` depending on whether the graph in question is valid. The algorithms actually implemented in funqual are slightly more complicated because they print helpful diagnostic messages to the user. Both sets of algorithms enforce the same rules, though.

4.3.1. Restrict Direct Call

$$restrict\_direct\_call(X, Y)$$

A restrict direct call rule creates a constraint that disallows functions with direct type X from calling functions with direct type Y. This constraint is relatively permissive because it still allows indirect calls from functions with direct type X to functions with direct type Y but is nonetheless checkable by this type system.

Figure 4.6 shows pseudocode for an algorithm that can check a call graph for violations of this rule. Assume that `edges` is a list of objects representing all the calls in the call graph.

This algorithm runs once per rule and terminates in linear time with respect to the number of edges in the call graph. To assert the correctness of this algorithm we will categorize each function call in this graph as one of two possibilities: a call to a standard function, or a call to a function pointer.

In the case of a standard function call, the correctness is trivial. The user must

```
function enforce_restrict_direct_call(X, Y, edges):
    for edge in edges:
        callee = edge.to
        caller = edge.from

        if X in caller.direct_type and Y in callee.direct_type:
            return false
    return true
```

Figure 4.6: Pseudocode for an algorithm that can check a *restrict_direct_call* constraint. This algorithm returns `true` if the call graph respects the constraint and `false` if the call graph violates it.

have annotated the direct type of both the caller and the callee[2]. If a function with direct type X calls a function with direct type Y, then `edges` will contain such an edge and in checking each edge we will detect it.

In the case of the function pointer call, we need to also examine all possible assignments of that function pointer. It is of course possible that the function pointer is null at runtime, but we will consider this type of error to be out of the scope of funqual. For the sake of this argument, let P stand for any function pointer and F stand for any function. For an assignment of F into P to be valid, F and P must have the same direct type. If they do not have the same direct type, then funqual will inform the user of an assignment type violation. If they do have the same direct type, then `edges` will contain an edge into P wherever P is called and that edge will be checked in the same way as a standard function call.

4.3.2. Restrict Indirect Call

$$restrict\_indirect\_call(X, Y)$$

[2]Funqual will check whatever was declared by the programmer — whether the programmer declared their intent correctly is outside the scope of this research.

```
function enforce_restrict_indirect_call(X, Y, edges):
    for edge in edges:
        callee = edge.to
        caller = edge.from

        if X in caller.direct_type and Y in callee.indirect_type:
            return false
    return true
```

Figure 4.7: Pseudocode for an algorithm that can check a *restrict_indirect_call* constraint. This algorithm returns `true` if the call graph respects the constraint and `false` if the call graph violates it.

A restrict indirect call rule creates a constraint that functions with direct type X cannot call functions with direct or indirect type Y. This has the effect of restricting functions with direct type X from calling functions with direct type Y, whether that call is direct or indirect. The need to enforce indirect calls in the presence of function pointers requires us to examine the indirect type of the callee for each edge.

Figure 4.7 shows pseudocode for an algorithm that can check a call graph for violations of this rule. Assume that `edges` is a list of objects representing all the calls in the call graph.

In order to simplify this algorithm, we will assume for the time being that indirect function types are inferred correctly. For an explanation of the indirect type inference algorithm and for an argument for its correctness, refer to Subsection 4.3.4. To assert the correctness of `enforce_restrict_indirect_call`, we will again consider each function call in the graph as a member of one of two categories: a call to a standard function, or an invocation of a function pointer.

In the case of a standard function call, the correctness is trivial. Assume function A with direct type X calls function B with indirect type Y. Since A directly calls B, we know that there will be an edge from A to B in the `edges` and when the algorithm

visits it, the algorithm will terminate with the claim that there is a violation.

In the case of a function pointer invocation, the rules of function pointer assignment come into play. If, via an invocation of B, a function of type Y could eventually be called, then the function pointer must necessarily have Y in its indirect type otherwise there would be an assignment error (for an in-depth argument of this refer to Subsection 4.3.4). As a result, when visiting the edge from A to B (where A is the function invoking function pointer B), the algorithm will detect that B has indirect type Y and will terminate with the claim that there is a violation.

4.3.3. Require Direct Call

$$require\_direct\_call(X, Y)$$

A require direct call rule creates a constraint that functions with direct type X can only call functions with direct type Y. Much like the restrict direct call rule, this rule is relatively easy to check and can be checked in time linear with respect to the number of edges in the call graph.

Figure 4.8 shows pseudocode for an algorithm that can check a call graph for violations of this rule. Assume that `edges` is a list of objects representing all the calls in the call graph.

To assert the correctness of this algorithm, we will categorize every function call as one of two possibilities: a call to a standard function, or a call to a function pointer.

In the case of a call to a standard function, the correctness is trivial. The user must have annotated the direct type of both the caller and the callee and we take these annotations to be correct. If a function with direct type X calls any function, then `edges` will contain an edge from the caller to the callee. Checking the direct types of caller and callee exhaustively for every edge in the graph will eventually find

```
function enforce_require_direct_call(X, Y, edges):
    for edge in edges:
        callee = edge.to
        caller = edge.from

        if X in caller.direct_type and Y not in callee.direct_type:
            return false
    return true
```

Figure 4.8: Pseudocode for an algorithm that can check a *require_direct_call* constraint. This algorithm returns `true` if the call graph respects the constraint and `false` if the call graph violates it.

any violations.

In the case of a function pointer call, we need to also examine all the possible assignments to that function pointer. Thankfully the assignment checker already checked the type safety of every function pointer assignment so we will assume that those are correct. In this case specifically, we can assume that, if the function which is actually called does not have direct type Y, then the function pointer which is called in code will also not have direct type Y. This call creates an edge which will certainly be visited by `enforce_restrict_direct_call` and so we can be certain that any function pointer invocation will be correctly checked in this regard.

4.3.4. Indirect Type Inference

While the user does not invoke indirect type inference in the same way that the user invokes the other rules, indirect type inference is still an important part of the type safety of funqual. This subsection explains indirect type inference and argues for the correctness of the algorithm.

Figure 4.9 shows pseudocode for an algorithm that can infer the indirect function

```
function infer_indirect_type(function, edges):
    indirect_types = empty set
    visited = empty set
    to_visit = empty set
    to_visit.add(function)

    while to_visit is not empty:
        curr = to_visit.pop()
        visited.add(curr)

        indirect_types.add_all(curr.direct_type)
        indirect_types.add_all(curr.indirect_type)

        for edge in edges:
            callee = edge.to
            caller = edge.from
            if caller == curr and callee not in visited:
                to_visit.add(callee)
    return indirect_types
```

Figure 4.9: Pseudocode for an algorithm to infer the indirect type of a function.

type for any function in the call graph. For the purpose of this function, we will let `function` be the function being checked. We will let `edges` be the list of edges in our graph and we will assume that it contains edges to function pointers where those function pointers are called. We also assume that `callee.indirect_type` is populated for function pointers but that it is an empty set for regular functions.

To assert the correctness of this algorithm imagine a function, F, from which evaluation eventually (either directly or indirectly) reaches a function, C, with type Y. We propose that because of the rules of this type system, it is necessary that Y is in the type of F. To demonstrate this we will break down the type pipeline into its

multiple cases.

The first case is that F calls C (either directly or indirectly) but that none of the calls from F to C are function pointer invocations. In this case, there will be a path in `edges` from F to C and because `infer_indirect_type` is a breadth first graph traversal starting at F, we know that the algorithm will eventually visit C. When the algorithm does visit C, it will grab the direct type of C (which contains Y) and add it to the indirect type of F. When the algorithm terminates, it will necessarily contain Y. In other words, if there is a path from F to C, the indirect type of F will contain the direct and indirect types of C.

The second case is that F invokes a function pointer P from which evaluation eventually results in a call to C. In this case, there may or may not be a path in `edges` from F to C. However, there will be a path in `edges` from F to P and an assignment of C into P. Recall that for an assignment of C into P to typecheck, the direct types of C and P must match and the indirect type of P must contain the indirect type of C. If Y is in the direct type of C, then Y must be in the direct type of P. Also if Y is in the indirect type of C, then Y must be in the indirect type of P. Since either the direct type or the indirect type of P must contain Y, we can reference case one and claim that because there is a path from F to P, and because the type of P contains Y, then Y will be in the indirect type of F.

The third case is an inductive step. Assume that F calls C but indirectly through some arbitrary number of function pointer invocations between. Let P_0 be a function pointer through which a call is made to C, let P_1 be a function pointer through which a call is made to P_0, let P_n be a function pointer through which a call is made to P_{n-1}, and let F call P_n. According to the logic in case two, if Y is in the direct type of C, then it must be in the direct type of P_0 or else the assignment will have failed. In the same way, if Y is in the direct or indirect type of P_{n-1}, then it must be in the

direct or indirect type of P_n. Inductively, Y must be in the direct or indirect type of P_n and because there is a path in `edges` from F to P_n, Y must end up in the indirect type of F. Lastly, as in case one, any of these calls (either from F to P_n, from P_n to P_{n-1}, or from P_0 to C) can be direct or indirect calls and Y will still be in the indirect type of F.

This algorithm terminates even in the presence of cycles because it tracks previously visited vertices in `visited` and does not visit them again. Even though these cyclic edges are not followed, the output is still correct because every vertex is visited once. Assume that F calls C and that C calls F. The algorithm first visits F, then visits C, but does not visit F again because F was added to `visited` when it was first examined. When F was added to `visited`, its direct and indirect type were added to the return value and the edges out of F were added to `to_visit`. All the necessary information was extracted from F on the first visit so visiting it again is not necessary.

4.4. Special Considerations when Creating a Call Graph

4.4.1. Dealing with Inheritance

When calling a virtual method in C++, it is impossible to know at compile time exactly which function is going to be run at run-time. This is very similar to the problem of function pointers (and in fact dynamic dispatch is usually implemented as a table of function pointers [16]) except that in the case of virtual functions we actually know statically the set of possible functions that could be called[3]. To account for this, we need to add extra edges to our call graph to represent all the possible places that a virtual method call could go.

[3]Funqual assumes that it has access to the full source code for call graph creation.

Let C be some function that calls $T.M$ where T is some class and M is a virtual method of T. When creating the call graph, we must surely add an edge from C to $T.M$. In addition to that, though, for any class S that is a subclass of T, we must also add an edge from C to $S.M$. This accounts for any possible overloads of M that might be called at run-time.

Figure 4.10 demonstrates this concept. It is a piece of C++ source code that calls a virtual function. Figure 4.11 shows the call graph for this code sample.

For this example we will continue to assume that there is a rule restricting indirect calls from `static_memory` functions to `dynamic_memory` functions. In `feedPanda` we can see that we call `Panda::Feed`. This is somewhat misleading: `Panda::Feed` is a virtual function and it is overridden by a child class called `RedPanda`. This means that any time `feedPanda` is called, it is impossible to know whether it is `Panda::Feed` being called or whether it is actually `RedPanda::Feed` being called. The only safe way to handle this scenario is to assume that `feedPanda` calls both of them. This is reflected in Figure 4.11 which is a call graph showing `feedPanda` pointing to both versions of the `Feed` function.

4.4.2. Overriding Methods with Annotations

Just like with the standard C++17 type qualifiers, if a virtual function $T.M$ is overridden by $S.M$, then the qualified types of $T.M$ and $S.M$ must match exactly. Figure 4.12 contains an example of an override that is invalid according to the C++17 standard. In this example, `Panda::Feed` has a `const` qualifier, but `RedPanda::Feed` does not. As such, the two functions have different types and the compiler will generate an error.

Funqual treats funqual direct type in the same way. For $T.M$ to be overridden by $S.M$, the two functions must have the same direct type. If they do not, funqual will

```
1  class Panda {
2  protected:
3      int m_hunger;
4  public:
5      virtual int Feed() {
6          m_hunger--;
7      }
8  };
9
10 class RedPanda : public Panda{
11 public:
12     int Feed() override {
13         Stomach *stomach = malloc(sizeof(Stomach));
14         memset(stomach, 0xFF, sizeof(Stomach));
15     }
16 };
17
18 void feedPanda(Panda *panda) static_memory {
19     panda->Feed();
20 }
21
22 int main(void) {
23     feedPanda(new RedPanda());
24 }
```

Figure 4.10: Example C++ program demonstrating inheritance. In `feedPanda`, it is impossible to know statically which instance of the `Feed` function will be called. Figure 4.11 shows the call graph for this program.

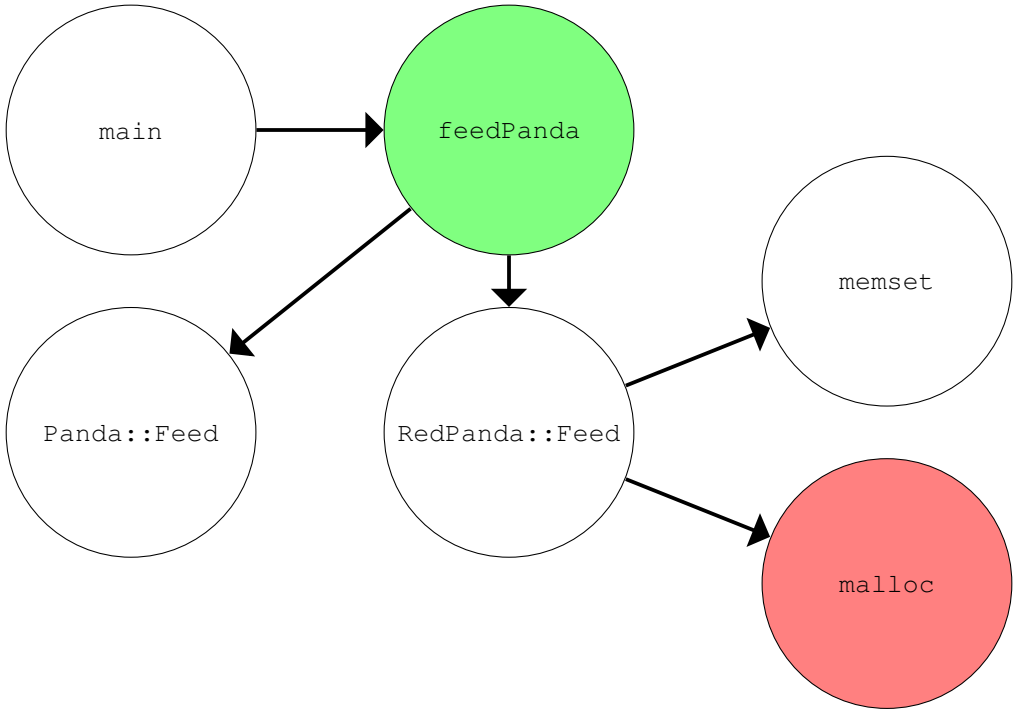

Figure 4.11: Call graph for Figure 4.10. Because `Panda::Feed` is a virtual function, we must draw an edge from `feedPanda` to every instance of `Feed`.

```
1 class Panda {
2 public:
3     virtual void Feed() const;
4 };
5
6 class RedPanda: public Panda {
7     virtual void Feed() override;
8 };
```

Figure 4.12: Example C++ containing an error. `Panda::Feed` and `RedPanda::Feed` have different types and so the override is invalid.

```
class Panda {
public:
    virtual void Feed() QTAG(static_memory);
};

class RedPanda: public Panda {
    virtual void Feed() override;
};
```

Figure 4.13: Example C++ containing a funqual type error. `Panda::Feed` and `RedPanda::Feed` have different types and so the override is invalid.

display an error. The Figure 4.13 shows a similar example of an invalid override but where the direct type is the type in conflict.

4.4.3. Operator Overloading

C++ allows for operator overloading. As a result, an expression such as `a = b + c;` could result in a function call depending on the types of `a` and `b`.

Compensating for this is relatively straightforward. When funqual comes across a binary or unary operator that can be overloaded, it checks the type of the operand(s) and checks for an operator overload. If there is an operator overload, then the call graph will contain an edge from the calling context to the overload function. If the overload is virtual, funqual checks for operator overloads in child classes as described in Section 4.4.1.

4.4.4. Bridging the Divide between Translation Units

The compilation of C++ code is driven by translation units. Translation units are the files which are provided to the C compiler to be translated into object files. In general, translation units are singular `.c` or `.cpp` files including any source files that

may be `#include`-ed. During this process, many symbols are said to have *external linkage* meaning that their type is specified in this translation unit but that their value is not (this is the case with extern variables, function prototypes, and class forward declarations). In these cases, examining the call graph of a single translation unit is not sufficient to enforcing global call graph constraints because we would not be able to see the calls made in other translation units which may be of interest for enforcing indirect call restrictions.

To solve this problem we need to examine every translation unit in the source and build a call graph that represents the entire codebase. In order to test this, we create several test cases where functions are defined in multiple translation units and where a function call graph constraint is violated between translation units.

Chapter 5

IMPLEMENTATION

This section contains information about the funqual tool including a discussion of how to use it, how it works, and what its limitations are.

5.1. Operation

Funqual is a tool that takes in C++ source code and a set of call graph rules and outputs a list of rule violations, if any exist. Section 5.1.1 demonstrates how to annotate C++ source code with funqual type qualifiers. Section 5.1.2 explains the syntax for writing down rules in the rules file. Section 5.1.3 shows the syntax for running funqual from the command line. Finally, Section 5.1.4 contains a few examples of programs, rules files, and the output from funqual.

5.1.1. Function Qualifier Annotations with QTAG and QTAG_IND

One of the goals of funqual was that it be entirely compatible with the C++17 standard. As such, funqual does not add any syntax to the language that would prevent annotated programs from being used by other tools (such as gcc or cppchecker). Additionally, any C++17 code that exists "in the wild" should be compatible with funqual with no modification. To this end, we use the existing C++17 annotation syntax to insert funqual type qualifiers.

For clarity and convenience we assume the following macros are in scope. These macros abbreviate the syntax for inserting the direct and indirect type qualifiers into program source. In practice, this macro can be inserted into the code alongside the annotations or can be placed in a utility library:

```
#ifndef FUNQUAL
#define FUNQUAL
// For direct type:
#define QTAG(TAG) \
    __attribute__((annotate("funqual::" #TAG)))
// For indirect type:
#define QTAG_IND(TAG) \
    __attribute__((annotate("funqual_indirect::" #TAG)))
#endif
```

Note that the __attribute__((annotate(foobar))) syntax is generally used for compiler-specific directives (like packed, align(8), noreturn, etc) and that attributes unknown by the compiler are simply ignored. This allows us to insert information into the AST that is available after parsing but which will not affect compilation.

Below is an example of the syntax for adding type qualifiers to a function. The function below has two qualified types: `static_memory` and `no_io`.

```
int main() QTAG(static_memory) QTAG(no_io) {
    return 0;
}
```

Below is an example of the syntax for adding type qualifiers to a method prototype within a class. The function below has qualified type `static_memory`.

```
class Panda {
    Panda() QTAG(static_memory);
};
```

Below is an example of the syntax for adding a type qualifier to a function pointer. The function pointer below has qualified type `static_memory`.

```
1 int QTAG(static_memory) (*func)(int, int);
```

Functions in the standard library can be annotated by simply repeating their prototype and adding a type qualifier annotation. During the first phase of type checking, funqual will scrape the entire codebase and determine the union of all type annotations for each function symbol. In the example below, `malloc` has two type qualifiers: `dynamic_memory` and `blocking`. Lines 1 and 3 could appear in the same file or in different files. There is no limit to the number of type qualifiers that can be applied to a function.

```
1 void *malloc(size_t size) QTAG(dynamic_memory);
2
3 void *malloc(size_t size) QTAG(blocking);
```

Function pointers must also be annotated with their indirect type. For a primer on the rules regarding indirect type and function pointer assignment, refer to Section 4.2. Below is an example of a function pointer with the indirect type `blocking`.

```
1 int QTAG_IND(blocking) (*func)(int, int);
```

5.1.2. Constrain the World! Writing a Rules File

Call graph rules are inserted into special files called rules files. By convention, rules files have the file extension `.qtag` but this convention is optional. Below is an example of a rules file that shows a few examples of each rule type:

```
1 rule restrict_indirect_call static_memory dynamic_memory
2 rule restrict_direct_call nonblocking blocking
3 rule require_direct_call nonblocking nonblocking
```

This rules file contains three rules: *restrict_indirect_call*(*static_memory*,

dynamic_memory), *restrict_direct_call*(*nonblocking*, *blocking*), and *require_direct_call*(*nonblocking*, *nonblocking*). As shown in this file, there is no process of declaring a type qualifier. They are brought into existence simply by referencing them.

In addition to specifying rules in a rules file, funqual also allows the user to specify additional function qualifiers in this file. In order to do this, the user must determine the clang Unified Symbol Resolution for the given symbol. This is a string that uniquely identifies the symbol across all translation units - it contains more information than the fully qualified name of the symbol because it needs to differentiate between static symbols in different translation units and it needs to differentiate between overloaded identifiers within the same translation unit. The Listing below demonstrates the syntax for adding the `dynamic_memory` qualifier to the stdlib `malloc`:

```
1 tag c:@F@malloc dynamic_memory
```

5.1.3. Running Funqual

Funqual can be run from the command line. There are two kinds of arguments: translation units and rules files. Arguments preceded by `-t` or `--tags-file` will be interpreted as a rules file. All other arguments will be interpreted as translation units. Funqual needs to be passed every translation unit in a project in order for it to create a representative call graph for the codebase. Below is an example command for running funqual. This command will pass in every `.cpp` file in the current directory and any subdirectories and will also pass in a rules file called `rules.qtag` in the current directory.

```
1 funqual *.cpp -t rules.qtag
```

5.1.4. Example Output

Below is the output of running funqual on Figure 4.10. Not only does funqual detect the presence of a rule violation, it also shows the exact sequence of calls that represent the violation. This information helps the user know that their code contains a type error and also helps the user to correct the error.

```
1 Rule violation: `dynamic_memory` function indirectly called from
    `static_memory` context
2         Path:   main.cpp::main() (38,5)
3         -calls: main.cpp::RedPanda::Feed(int) (31,18)
4         -calls: main.cpp::(#include)::malloc(size_t) (466,14)
```

Funqual will, by default, output every illegal path in a program. This has the potential to generate a lot of output for larger programs with many violations.

5.2. Practical Limitations

Because of complexities in parsing C++, certain applications of function pointers are not currently checkable by funqual. Specifically, any expression where the lvalue or rvalue in a function pointer assignment is anything other than a raw variable can not be checked. Figure 5.1 shows a few examples of assignment expressions that funqual cannot check correctly.

The difficulty here arises from determining the type of an expression using the libClang API. Types containing function pointers can be annotated by inserting `QTAG` and `QTAG_IND` annotations. Additionally, the type of any expression in the libClang AST can be queried. However, the type returned by libClang when querying for the type of an expression will not contain type qualifier annotations. In order to get the type annotations for lvalues and rvalues in an assignment, funqual needs to look

```
1  void *(**array)(size_t size) QTAG(dynamic_memory);
2
3  //assignment not checkable because the array dereference in lvalue
4  array[0] = malloc;
5
6  struct {
7      void *(*field)(size_t size);
8  } structure;
9
10 struct structure s;
11
12 //assignment not checkable because of field dereference in lvalue
13 s.field = malloc;
14
15 void *(**arr)(size_t);
16
17 //assignment not checkable because of array dereference in rvalue
18 arr = array[0];
```

Figure 5.1: Examples of function pointer assignment expressions that are not checked correctly by funqual

up the declaration of the identifier and parse it. Types in C++ can be arbitrarily nested and function pointers can be hidden within complicated types. In order to limit complexity of the funqual tool, a decision was made that funqual would only support assignments to and from raw identifiers.

For the same reason, funqual does not support casting function types to coerce them into having certain type qualifiers. When determining the type of the result of a cast expression, libClang ignores the type qualifier annotations and so the cast expression loses that information after parsing.

Chapter 6

APPLICATION

This chapter demonstrates three real use-cases for funqual. In each section below, we outline the application of funqual to a project, the constraint that funqual was used to enforce, and the outcome of using funqual to check that constraint. Section 6.1 describes using funqual to prevent reentrancy errors in a class assignment for Operating Systems at Cal Poly (CSC453). Section 6.2 describes using funqual to prevent use of `malloc` and `printf` during boot-up of a custom kernel written in a class assignment for Operating Systems 2 at Cal Poly (CSC454). Lastly, Section 6.3 describes using funqual to prevent the use of potentially blocking calls in high frequency loops in a robotics application.

All of these projects were developed before funqual existed, so funqual was not used during the development cycle. The goal of this chapter is to demonstrate that funqual can scale beyond small test cases and to demonstrate how funqual can be used to address a variety of real-world issues.

6.1. Glibc Nonreentrant Functions

The GNU C Library Reference Manual warns against calling nonreentrant functions from signal handlers [10]. A function which only accesses memory within its stack frame is reentrant because it cannot be affected by external state. A function which accesses heap, global, or static memory may be nonreentrant if that memory can be modified by other execution environments. This includes functions which reference a global datastructure (e.g. `malloc`) or grab a global lock (e.g. `printf`). Reentrancy is a separate but similar concept to thread-safety; a reentrant function is thread-safe but

a thread-safe function may not necessarily be reentrant. As an example, `printf` could be considered thread safe because it locks the stream while writing to it. However, if a call to `printf` is interrupted while it holds the lock and the interrupt handler makes its own call to `printf`, then the interrupt handler will wait for the lock. Since the code holding the lock cannot run until the interrupt handler finishes, the system is in deadlock. This is bad — we would like to prevent this error as well as errors like it.

Funqual can find and report this type of error. To demonstrate this, we take a class assignment written for an Operating Systems class (CSC453) that uses signal handlers, insert function type qualifiers, and create a rules file. The assignment was to simulate a set of snakes crawling around the screen. Each time the user presses control-C (creating a `SIGINT` signal), one of the snakes disappears. When the user tries to kill the process (creating a `SIGQUIT` signal), the program makes each snake disappear and then terminates. If a signal is sent during a call to a nonreentrant function, that function is preempted by the signal handler; if the signal handler calls that same nonreentrant function, this can result in undefined behavior.

To make funqual detect this issue, we use two type qualifiers: `preemptive` which applies to signal handlers, and `non_reentrant` which applies to nonreentrant functions. We also create one rule: $restrict\_indirect\_call(preemptive, non\_reentrant)$. Since many of the nonreentrant functions we are concerned about are in the C standard library, these functions are annotated as `non_reentrant` in the rules file. Figure 6.1 shows the rules file used. The list of functions tagged as nonreentrant is incomplete but represents the ones used in this program. In addition to tagging nonreentrant library functions in the rules file, the signal handlers in the code are tagged as `preemptive`. Figure 6.2 shows the two lines that were added to the program source to tag signal handlers.

```
rule restrict_indirect_call preemptive non_reentrant

tag c:@F@malloc non_reentrant
tag c:@F@free non_reentrant
tag c:@F@printf non_reentrant
tag c:@F@fprintf non_reentrant
tag c:@F@sprintf non_reentrant
tag c:@F@rand non_reentrant
```

Figure 6.1: Rules file for preventing preemptive functions from calling non_reentrant functions. Since this rules file contains no references to project-specific functions, the file could conceivably be re-used by several projects.

```
void kill_snake() QTAG(preemptive);
void lwp_stop() QTAG(preemptive);
```

Figure 6.2: Lines inserted into C file to mark signal handlers as preemptive.

The size of this project was 458 lines of code[1] contained in five `.c` files with 40 edges in the call graph. Funqual analyzed the source in about 0.9 seconds[2] — 0.1 seconds were spent in libClang parsing the source, 0.8 seconds were spent traversing the AST to generate the call graph, 0.001 seconds were spent performing type inference, 0.000,01 seconds were spent checking the call graph, and 0.000,01 seconds were spent checking assignments.

On the first run, funqual did not detect any call graph violations. In order to test that the tool does actually detect errors, several illicit calls to `printf` were inserted. After doing so, funqual correctly detected and reported these errors. Figure **??** shows the output from funqual when run on this modified codebase.

[1]Line count achieved using the `cloc` utility not including comments or blank lines.
[2]Data collected on a T460 Lenovo Thinkpad with Quad Intel Core i5-6300U CPU at 2.4GHz.

6.2. Restricting API available during kernel initialization

Kernel development is complicated for a variety of reasons. One reason that makes it particularly complicated is that not all of the standard libraries are available from certain contexts in the kernel. Just like in Section 6.1, it would be a serious issue if we were to call `malloc` or `printf` from within an interrupt handler. In addition to that, we also need to ensure that these functions are not called before their associated interfaces are initialized. As an example, the use of `malloc` depends on the page table having been initialized, and so there are contexts within kernel initialization where a call to `malloc` would be inappropriate. We would like to prevent this error as well as errors like it.

To demonstrate how funqual can be applied to this problem, we take an assignment for Operating Systems II (CSC454) where students develop their own simple x86_64 kernel, and augment it with function type qualifiers and a rules file. The kernel consists of several subsystems that are each initialized in sequence. These subsystems, in order, are: a VGA subsystem to display text on the screen, a PC2 subsystem to poll the keyboard and mouse for input, a subsystem to schedule interrupts and register interrupt handlers, an interface to send and receive text over the system's serial interface, a memory manager to allocate physical pages and add them to the page table, a scheduler to run multiple processes, and lastly a set of processes which each draw a snake crawling around the screen like in Section 6.1. These subsystems all depend on things in the previous subsystems. The keyboard subsystem depends on the VGA subsystem to display the keys the user pressed, the physical memory manager depends on the interrupt subsystem to listen for page faults, `malloc` depends on the memory manager, and so on. It is very easy in the early stages of boot-up to accidentally call a function in a subsystem that has not been initialized. Sometimes these calls are hidden by a few edges in the call graph, making it hard for a human

to detect them.

To solve this problem, we use several type qualifiers: `pre_vga`, `vga`, `pre_pc2`, `pc2`, `pre_irq`, `irq`, `pre_ser`, `ser`, `pre_mmu`, `mmu`, `pre_proc`, and `proc`. We also create rules for each subsystem that prevent functions tagged `pre_XXX` from calling functions in any of the subsystems that involve `XXX`. Figure 6.4 shows the rules file used to support this.

The subsystems are initialized in the following order: VGA, PC2, Interrupt Request (IRQ), Serial, Memory Management Unit (MMU), and process manager (proc). For each subsystem, there is an `_init` function which is called to set the subsystem up. The `_init` function for each subsystem is annotated with `pre_XXX` for each subsystem that it precedes. Figure 6.5 shows the annotations that were added to accomplish this.

```
rule restrict_indirect_call pre_vga vga
rule restrict_indirect_call pre_vga pc2
rule restrict_indirect_call pre_vga irq
rule restrict_indirect_call pre_vga ser
rule restrict_indirect_call pre_vga mmu
rule restrict_indirect_call pre_vga proc

rule restrict_indirect_call pre_pc2 pc2
rule restrict_indirect_call pre_pc2 irq
rule restrict_indirect_call pre_pc2 ser
rule restrict_indirect_call pre_pc2 mmu
rule restrict_indirect_call pre_pc2 proc

rule restrict_indirect_call pre_irq irq
rule restrict_indirect_call pre_irq ser
rule restrict_indirect_call pre_irq mmu
rule restrict_indirect_call pre_irq proc

rule restrict_indirect_call pre_ser ser
rule restrict_indirect_call pre_ser mmu
rule restrict_indirect_call pre_ser proc

rule restrict_indirect_call pre_mmu mmu
rule restrict_indirect_call pre_mmu proc

rule restrict_indirect_call pre_proc proc
```

Figure 6.4: Rules file for a simple kernel written for CSC454. The rules written here are intended to prevent code which runs before a subsystem is initialized from calling any function that depend on the subsystem.

```
1  // for VGA subsystem
2  bool VGA_init() QTAG(pre_pc2);
3  void set_char_at(int row, int col, char character, char attributes)
4      QTAG(vga) QTAG(pre_pc2);
5
6  // for PC2 subsystem
7  void PC2_init() QTAG(pre_irq);
8  bool get_char(char *ret) QTAG(pc2) QTAG(pre_irq);
9
10 // for interrupt request subsystem
11 void IRQ_init() QTAG(pre_ser);
12 void IRQ_set_handler(int irq, irq_handler_t handler, void *args)
13     QTAG(irq) QTAG(pre_ser);
14
15 // for serial subsystem
16 void SER_init() QTAG(pre_mmu);
17 int SER_write(const char *buff, int len) QTAG(ser) QTAG(pre_mmu);
18
19 // for memory management unit subsystem
20 void MMU_pt_init() QTAG(pre_proc);
21 void *MMU_alloc_page() QTAG(mmu) QTAG(pre_proc);
22 void MMU_free_page(void *page) QTAG(mmu) QTAG(pre_proc);
23
24 // for multiprocessing subsystem
25 void PROC_init();
26 Process_t *PROC_create_kthread(kproc_t entry_point, void *arg)
27     QTAG(proc);
28 void PROC_run() QTAG(proc);
```

Figure 6.5: Lines inserted into C source for a simple kernel in order to prevent subsystems from depending on interfaces not yet initialized.

This project contained 6034 lines of C code spread over 36 files with 291 edges in the call graph[3]. Funqual analyzed this source in about 2.2 seconds — 0.3 seconds were spend in libClang parsing source files, 1.9 seconds were spent building the call graph, 0.005 seconds were spent performing type inference, less than 0.000,05 seconds were spent checking the call graph, and less than 0.000,05 seconds were spent checking function pointer assignments.

Funqual did detect errors in this source. Figure 6.6 shows the output from running funqual. According to this output, funqual detected several cases where a `pre_ser` function called a `ser` function. Specifically, `IRQ_init` called `printk` which eventually calls `SER_write`. `printk` in this project operates just like `printf`. The only difference is that `printk` outputs to both VGA and to serial. This rule violation represents an actual error that existed in the project which was not detected until funqual found it. Ideally, `SER_write` just fills a buffer; this buffer is emptied when the serial port sends an interrupt requesting data. However, if the serial interface isn't operating, that buffer might fill up causing the caller to block until space becomes available in the buffer. If we had filled the entire 1024 character buffer before initializing the serial interface, the kernel would have blocked permanently.

Observing the version history of this codebase tells the full story of how this bug occurred. Initially, `printk` only printed to VGA. At that point in time, it was okay to call `printk` from `IRQ_init` because `printk` only depended on VGA. However, at some point a decision was made that `printk` should print to both VGA and serial. At this point, a simple modification was made to `print_str` without considering all the ways that `printk` was used.

This sort of mistake is probably very familiar to many programmers and it might exist for quite a while before presenting as an error. Imagine what the symptoms

[3]Line count achieved using the `cloc` utility not including comments or blank lines. Many of these lines were machine-generated.

```
Rule violation: `ser` function indirectly called from `pre_ser` context
        Path:   src/IRQ.c::IRQ_init() (12,6)
        -calls: src/io.c::printk(const char *, ...) (7,5)
        -calls: src/io.c::print_char(char) (9,6)
        -calls: src/main.c::SER_write(const char *, int) (7,5)

Rule violation: `ser` function indirectly called from `pre_ser` context
        Path:   src/IRQ.c::IRQ_init() (12,6)
        -calls: src/io.c::printk(const char *, ...) (7,5)
        -calls: src/io.c::print_str(const char *) (10,6)
        -calls: src/main.c::SER_write(const char *, int) (7,5)

Rule violation: `ser` function indirectly called from `pre_ser` context
        Path:   src/IRQ.c::IRQ_init() (12,6)
        -calls: src/io.c::printk(const char *, ...) (7,5)
        -calls: src/io.c::print_long_hex(uint64_t, bool) (13,6)
        -calls: src/io.c::print_str(const char *) (10,6)
        -calls: src/main.c::SER_write(const char *, int) (7,5)

Rule violation: `ser` function indirectly called from `pre_ser` context
        Path:   src/IRQ.c::IRQ_init() (12,6)
        -calls: src/io.c::printk(const char *, ...) (7,5)
        -calls: src/io.c::print_decimal(int64_t) (14,6)
        -calls: src/io.c::print_str(const char *) (10,6)
        -calls: src/main.c::SER_write(const char *, int) (7,5)
```

Figure 6.6: Output from funqual when run on simple OS kernel

would have been: the programmer would have added an extra `printk` to `IRQ_init` and the programmer would have observed that the program hanged. Why did it hang, though? In kernel development, there are a lot of things that could cause a hang, but a simple call to `printk` might not be the first thing one would check. It would take quite a lot of digging to go from `printk` to `print_char` to `SER_write` to then see that the issue was there.

This situation really demonstrates the strength of funqual. The programmer cannot possibly be cognizant of every location from which a function is called. What's more, the programmer is usually unaware of paths between functions, especially when there are several levels of indirection in-between them. Letting the programmer express their intuition as hard-coded rules and using a tool to check those rules automatically enables the programmer to be confident that errors like this don't occur in practice.

6.3. Detecting slow function calls in high frequency contexts

In robotic motion control, proper timing is paramount to good performance. In industrial automation, control loops[4] often run on special hardware with real-time guarantees at several thousand cycles per second. Specialized hardware like this is out of reach for robotics hobbyists, so we use general purpose hardware and GNU/Linux to achieve similar results.

Without the real-time guarantees of a specialized environment, it is very difficult to maintain consistent control loops even at 100Hz. Part of the problem is that the GNU/Linux "real-time" scheduler is subject to some minor jitter [2], but a lot of the

[4]A control loop is an algorithm that measures some aspect of a system, calculates an output vector, and applies that output vector in a continuous loop until the system reaches a desired state. For example, a car's cruise-control will measure the current speed, calculate how much gas to apply, and apply that much gas in a tight loop. For each iteration, the process of taking input, calculating output, and applying output is called one *cycle*.

problem stems from the fact that some standard library functions are just too slow to run in these high frequency contexts. Over years of troubleshooting slow control loops, the software team at the Atascadero Education Foundation has created a list of functions which are sometimes slow and should never be called in these high frequency contexts. We would like a tool to statically check that these high frequency contexts never call into functions in this list of slow functions.

We use funqual to find and report these issues. To do this, we use two type qualifiers: `hi_freq` to represent these high-frequency control loops, and `slow` to represent those functions which should not be called from within a control loop. We also create one rule: $restrict\_indirect\_call(hi\_freq, slow)$ which tells funqual that `hi_freq` functions should never call `slow` functions whether directly or indirectly.

An alternative approach which we abandoned because of increased burden on the programmer is to use just one function qualifier, `fast`, and one rule, $require\_direct\_call($ $fast, fast)$, to require that each function marked `fast` only call other functions marked `fast`. The issue with this approach is that functions would be guilty until proven innocent — there are hundreds of library functions that we do use in control loops without issue and each of these would need to be marked as `fast` before funqual would accept them. This increases the adoption cost for this approach and requires a lot of annotation. If we were more serious about guaranteeing close to real-time performance, this approach would have been more appealing.

Figure 6.7 shows the rules file that was written to accomplish this. Several functions from various libraries are tagged in the rules file as `slow` and there is one rule restricting `hi_freq` functions from calling `slow` functions. This is not an exhaustive list of slow functions, but it is a representative subset of them.

In addition to creating the rules file which contains the rule and which marks several functions as `slow`, we also modified several lines in the source to mark certain

```
rule restrict_indirect_call hi_freq slow

tag c:@F@malloc slow
tag c:@F@printf slow
tag c:@F@fprintf slow
tag c:@F@sprintf slow
tag c:@N@frc@S@CameraServer@F@GetInstance#S slow
tag c:@N@frc@S@DriverStation@F@GetInstance#S slow
tag c:@N@frc@S@Scheduler@F@GetInstance#S slow
tag c:@N@nt@S@NetworkTableEntry@F@GetInstance#1 slow
tag c:@N@nt@S@NetworkTable@F@GetInstance#1 slow
```

Figure 6.7: Rules file for preventing high frequency functions from calling slow functions. Several functions from standard libraries are marked in the rules file as `slow`.

functions as `hi_freq`. Figure 6.8 shows the lines that were added to the C++ source file in order to achieve this. This is not an exhaustive list of high frequency functions, but it is a representative subset of them.

This codebase is big, in large part because of all the libraries we depend on for interfacing with sensors and actuators on the robot. In the interest of time we could not check the entire codebase but rather we focused on the core libraries and the subsystems containing control loops. The portion of the library that we checked consists of 6959 lines of C++ code spread out over 42 files. The header files we include from other libraries consists of 12,506 lines of code spread out over 145 files. As such, analyzing these files takes a long time. Funqual analyzed the source in about 4 minutes — 24 seconds were spent in libClang parsing the source, 209 seconds were spent traversing the AST building a call graph, 0.04 seconds were spent performing type inference, 0.000,02 seconds were spend checking the call graph, and less than 0.000,005 seconds were spent checking function pointer assignments. The call graph contains 11635 vertices and 5103 edges. Obviously due to the size of this project, it

```
1 void CoopMTRobot::DisabledPeriodic() QTAG(hi_freq) override;
2 void CoopMTRobot::AutonomousPeriodic() QTAG(hi_freq) override;
3 void CoopMTRobot::TeleopPeriodic() QTAG(hi_freq) override;
4 void CoopMTRobot::TestPeriodic() QTAG(hi_freq) override;
5
6 static void *SPIGyro::Run() QTAG(hi_freq);
7
8 bool SmartPixy::getStart() QTAG(hi_freq);
```

Figure 6.8: Lines inserted into C++ source file to mark certain functions as `hi_freq`.

takes a long time for funqual to traverse it all. Usually, projects using clang alleviate this by doing incremental compilation — only the files which changed need to be examined. Funqual does not currently support incremental linting (implementing it is certainly possible but would take significant development time). If funqual did support incremental linting, then the time to run funqual on the codebase would be significantly reduced for most runs.

Funqual found many errors in this codebase. Because the funqual output is so large, the entirety is not included here, but a representative portion of the output is shown in Figure 6.9. Most of these errors relate to stray debug `printf`s inserted into the code.

These calls were inserted for temporary debugging purposes and should definitely be removed. Calls to `printf` sometimes block for up to a few milliseconds when the output buffer gets filled and data needs to be copied somewhere else. When a loop runs at 100Hz (10ms per cycle), a delay of a few milliseconds can slow the loop and degrade performance. As such, these rule violations represent actual errors in the source code which were found using funqual.

```
1  Rule violation: `slow` function indirectly called from `hi_freq` context
2          Path:   lib/sensors/SPIGyro.cpp::frc973::SPIGyro::Run(void *)
       (60,18)
3          -calls: lib/sensors/SPIGyro.cpp::frc973::SPIGyro::ReadPartID()
       (134,14)
4          -calls:
       lib/sensors/SPIGyro.cpp::frc973::SPIGyro::DoRead(uint8_t) (92,14)
5          -calls:
       lib/sensors/SPIGyro.cpp::frc973::SPIGyro::DoTransaction(uint32_t,
       uint32_t *) (128,10)
6          -calls: stdio.h::printf(const char *__restrict, ...) (362,12)
7
8  Rule violation: `slow` function indirectly called from `hi_freq` context
9          Path:   lib/sensors/SPIGyro.cpp::frc973::SPIGyro::Run(void *)
       (60,18)
10         -calls:
       lib/sensors/SPIGyro.cpp::frc973::SPIGyro::InitializeGyro() (67,10)
11         -calls: stdio.h)::printf(const char *__restrict, ...) (362,12)
12
13 Rule violation: `slow` function indirectly called from `hi_freq` context
14         Path:   lib/sensors/SPIGyro.cpp::frc973::SPIGyro::Run(void *)
       (60,18)
15         -calls:
       lib/sensors/SPIGyro.cpp::frc973::SPIGyro::InitializeGyro() (67,10)
16         -calls:
       lib/sensors/SPIGyro.cpp::frc973::SPIGyro::DoTransaction(uint32_t,
       uint32_t *) (128,10)
17         -calls: stdio.h::printf(const char *__restrict, ...) (362,12)
```

Figure 6.9: Output of running funqual on robotics library. This is not the entire output, but rather a small snippet of it

Chapter 7

FUTURE WORK

Funqual is only a proof-of-concept and an exploration of user-defined call graph constraints. As such, it leaves a lot of work open for further exploration. This future work generally falls into three categories: expanding the abilities of the funqual tool, researching the impact that a tool like funqual could have during the development cycle, and expanding the type system discussed in this thesis.

Funqual is unable to correctly check every program. The most striking issue is the ability to check typedefs, array members, or struct members whose types are annotated function pointers. Given the current libClang api, type checking these more complicated expressions is difficult. At the moment, when querying an expression in the clang AST to determine the expression's type, attributes that were in the declaration are not included. If these attributes were included, it would make querying the funqual type of any arbitrary expression trivial since the funqual type is encoded as an attribute. Future work could go into improving the implementation of funqual as well as the libClang api.

This research lacks any form of usability testing. Funqual as a tool exists, and it can check programs against arbitrary constraints, but we currently have no idea how useful it is. How often do developers need to check constraints like these? How easy is funqual for developers to use? Does a tool like funqual actually help developers while they are developing software? How do we teach developers to think about the call graph and about how to restrict it? Static Analysis tools exist to assist the developer, so in order to apply funqual to real world projects, all these questions must be answered. A good deal of future work could go into answering these questions and into determining how we measure the value of a tool like this.

The type system described in this thesis supports three types of constraints: *require_direct_call*, *restrict_direct_call*, and *restrict_indirect_call*. There are without a doubt many other rules that could be implemented. For example, we might want to restrict the maximum stack depth reachable from a function (i.e. limit the depth of the call graph that is reachable from a function as well as prevent recursion) for situations where we need to limit how much stack space is used (e.g., when writing an interrupt service routine that runs on a fixed 1Kb interrupt stack). Additionally, funqual is set up to accept whatever input it is given from the user. It might be possible to infer certain things about the direct type of functions based on their content or usage. For example, a function which references global, static, or heap memory may be inferred to be nonreentrant; or a function that is registered as an interrupt handler may be inferred to be preemptive. This would significantly reduce the load on the programmer to insert these annotations manually. Any and all additions to this type system just makes funqual and the concept of call graph constraint more useful.

Clearly there is a lot of work that could be done on funqual. The idea that is brought to life in this thesis is in its early infancy and needs to mature before it is ready to compete with other methods of static analysis. More features need to be added to the tool, metrics need to be created so that funqual can be properly compared to other tools in its class, and the type system could be expanded to make it more usable. A rich body of research could easily find its foundation herein.

Chapter 8

CONCLUSION

In the beginning of this thesis, we demonstrated a certain type of error in C++ code. The specific example was of `printf`, a non-reentrant function, being called from inside a signal handler. This one simple problem was expanded to represent a whole class of issues which were easy to describe but difficult to check using existing methods. To solve this problem, we created a type system and a tool to enforce that type system.

Funqual turns program source into a directed call graph and gives the user a syntax with which to encode their own constraints into this call graph. In this call graph, functions are represented as vertices and function calls are represented as edges. The constraints describe which functions are allowed to call which other functions, or more abstractly, which vertices in the call graph are allowed to have paths to which other vertices. Given this description of the problem, determining whether a program follows these type constraints is algorithmically simple.

Chapter 4 formalizes these concepts. Section 4.2 describes rules for how function pointers are represented in this graph. Section 4.4 describes how special cases like inheritance and operator overloading are represented in this graph. Finally Section 4.3 formally describes the rules of the type system and provides some semi-formal arguments that the type system described here is sound.

Chapter 6 takes funqual and applies it to three actual C++ projects to enforce realistic constraints. The first application was a small program which demonstrates funqual's ability to detect re-entrancy errors in code that contains signal handlers. This project did not originally contain any issues but funqual was able to detect errors that were manually inserted. The second application was a small operating

system kernel written for a class at Cal Poly which demonstrates funqual's ability to detect calls to functions before their associated subsystems were initialized. Funqual found actual errors in this project that the author did not know about before analysis. The third application was a robotics library with soft real-time requirements which demonstrates funqual's ability to detect inappropriate calls to slow functions. Funqual again found actual errors in this project that the authors did not know about before analysis. All three of these projects had different domain constraints that funqual was used to enforce — the fact that funqual can correctly analyze and detect a variety of issues shows the versatility of this tool.

Finally, Chapter 7 describes all the future research topics that could be pursued via call graph constraint checking. Some of this future work revolves around improving the tool funqual itself, some of this future work revolves around expanding the scope of the type system, and some of this future work revolves around measuring the impact that call graph constraints have on the development process. There are a lot of open questions in this area. Funqual has been demonstrated to detect realistic bugs in non-trivial projects; several of these bugs were unknown to the project authors before analysis. This shows that the methods described here have considerable potential, and that incorporating a tool like funqual into the project development cycle might significantly reduce the occurrence of these types of errors. Measuring this impact should be a focal point of future work.

Static analysis is a fun and interesting topic for research but it is not a purely academic pursuit. The goal of funqual, and tools like it, is to help people write high quality software. No single tool can possibly achieve this on its own, but with new tools and new techniques more and more issues can be detected. Funqual on its own may not have a huge impact on the development cycle, but at this moment the ideas behind funqual are ready to be incorporated into the existing pantheon of C++ static analysis tools.

BIBLIOGRAPHY

[1] Static analysis. https://wiki.mozilla.org/Static_Analysis, November 2017.

[2] V. Bridgers. Real time linux scheduling comparison. https://elinux.org/images/d/de/Real_Time_Linux_Scheduling_Performance_Comparison.pdf, 2015.

[3] D. Evans. Static detection of dynamic memory errors. *SIGPLAN*, May 1996.

[4] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. *SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[5] J. Foster, M. Faehnlich, and A. Aiken. A theory of type qualifiers. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, May 1999.

[6] T. Glek. Dehydra, prcheck, squash in mercurial. https://blog.mozilla.org/tglek/2007/07/13/dehydra-prcheck-squash-in-mercurial/, July 2007.

[7] D. Greenfieldboyce and J. Foster. Type qualifiers for java. August 2005.

[8] D. Greenfieldboyce and J. Foster. Type qualifier inference for java. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007.

[9] Y. Kreinin. Defective c++. http://www.yosefk.com/c++fqa/defective.html, November 2016.

[10] S. Loosemore, R. M. Stallman, and A. Oram. The gnu c library reference manual.

[11] N. Matsakis and F. S. K. II. The rust language. *ACM SIGAda*, October 2014.

[12] H. Ogasawara, M. Aizawa, and A. Yamada. Experiences with program static analysis. *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, 1998.

[13] V. Ordy. Writing and designing c++ extensions and transformers. December 2009.

[14] Y. Padioleau. Parsing c/c++ code without pre-processing. In *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] S. Schaub and B. A. Malloy. Comprehensive analysis of c++ applications using libclang api. October 2014.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[17] T. R. Team. The rust programming language. May 2015.

[18] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006.