FUNQUAL: USER-DEFINED, STATICALLY-CHECKED CALL-TREE 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-Tree 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-Tree 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 tree based on these type qualifiers. We demonstrate that this tool, when used with hand-crafted 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, geniusly strict, and idiotically sloppy. C++17 is a hot mess but it's everwhere - 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 [15]. 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. All these tools have different use cases and can be used in conjunction 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[15], some tools examine ownership of objects to enforce memory safety[9], some tools examine the flow of values through a program to ensure security[7], and many other tools do other things entirely.

This paper intends to add a new technique to the existing arsenal making it possible to check for errors which were previously undetectable. To motivate this technique, we provide a problematic example. The following snippet of C code has a bug in it - the reader is implored 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);
    }
}
```

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.

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 difficuly for code at scale. Let's say, for instnace, 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 judgement that occured 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 [4]. 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 [4]. 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.

```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");
  }
}
```

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

As demonstrated in Figure 2.1, 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 pointing from node X to node Y in the associated call graph.

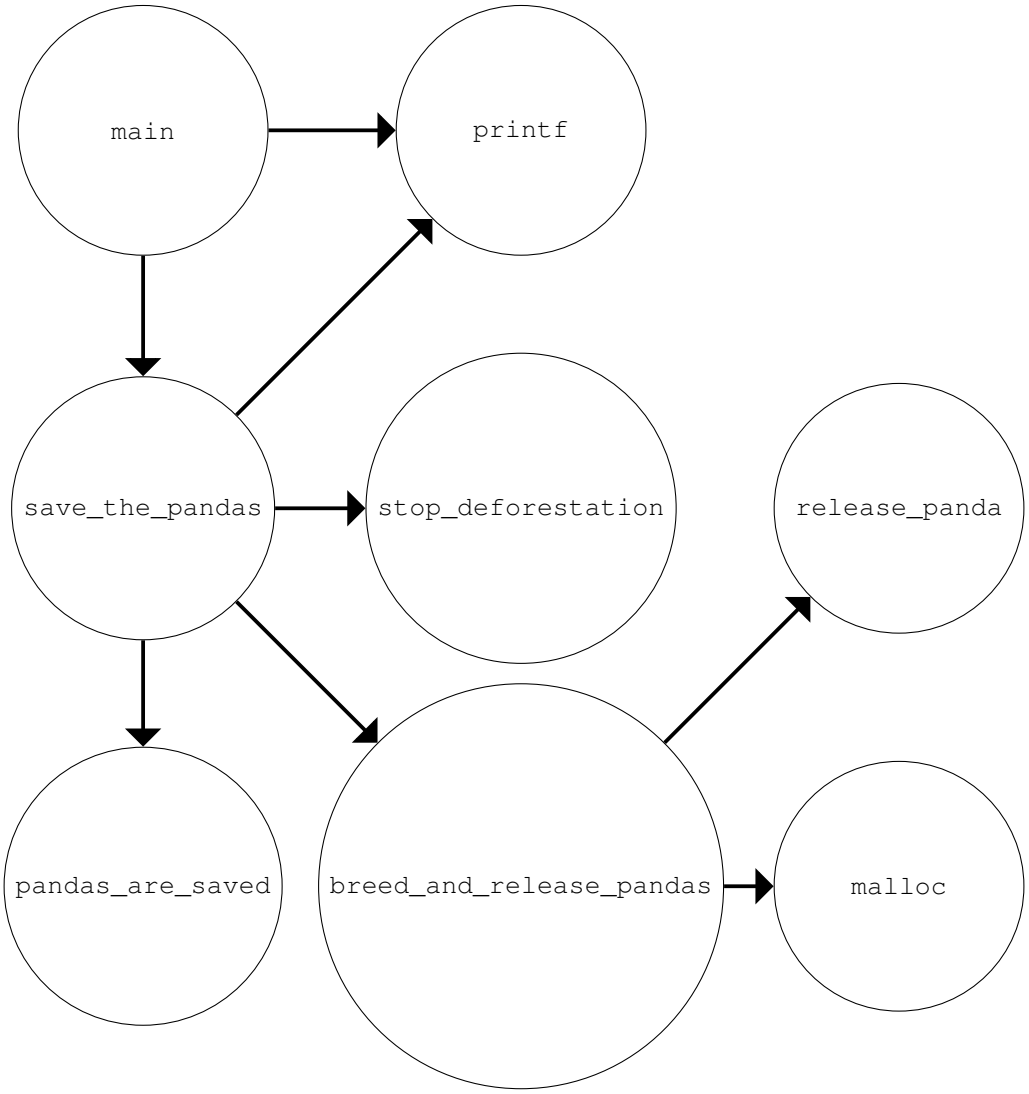

Figure 2.1: Example Call Graph. The source code associated with this call graph is showin in Listing 2.1

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 what functions call what 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 publishes several journals that 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 this research to happen.

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 [15, 10].

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 Mozillla'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 [5, 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" [9]. 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 [14]. 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 [3].

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 [2]. 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 program 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 [4].

"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 [4]. 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. 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 [4]. 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 [8, 13, 11, 12]. Years of language additions, the need for backwards compatibility, and the existence of a text-based preprocessor 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 [13]. 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 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 [13]. 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 script 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 4.3 is a detailed explanation of each of the call graph constraints supported by funqual. Section 4.2 explains how type qualifiers are applied to function pointers and how funqual checks them. Finally, Section 4.4 explains a few special cases and explains how funqual handles them to create a complete call graph.

Note that this Section 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 the Background Section. It is reproduced in this Section as Listing 4.1 for convenience.

```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");
  }
}
```

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

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 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. To accomplish this we will create two type qualifiers: `static_memory` and `dynamic_memory`. We will 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. 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 4.1 shows the call graph for Listing 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 safety 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 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.

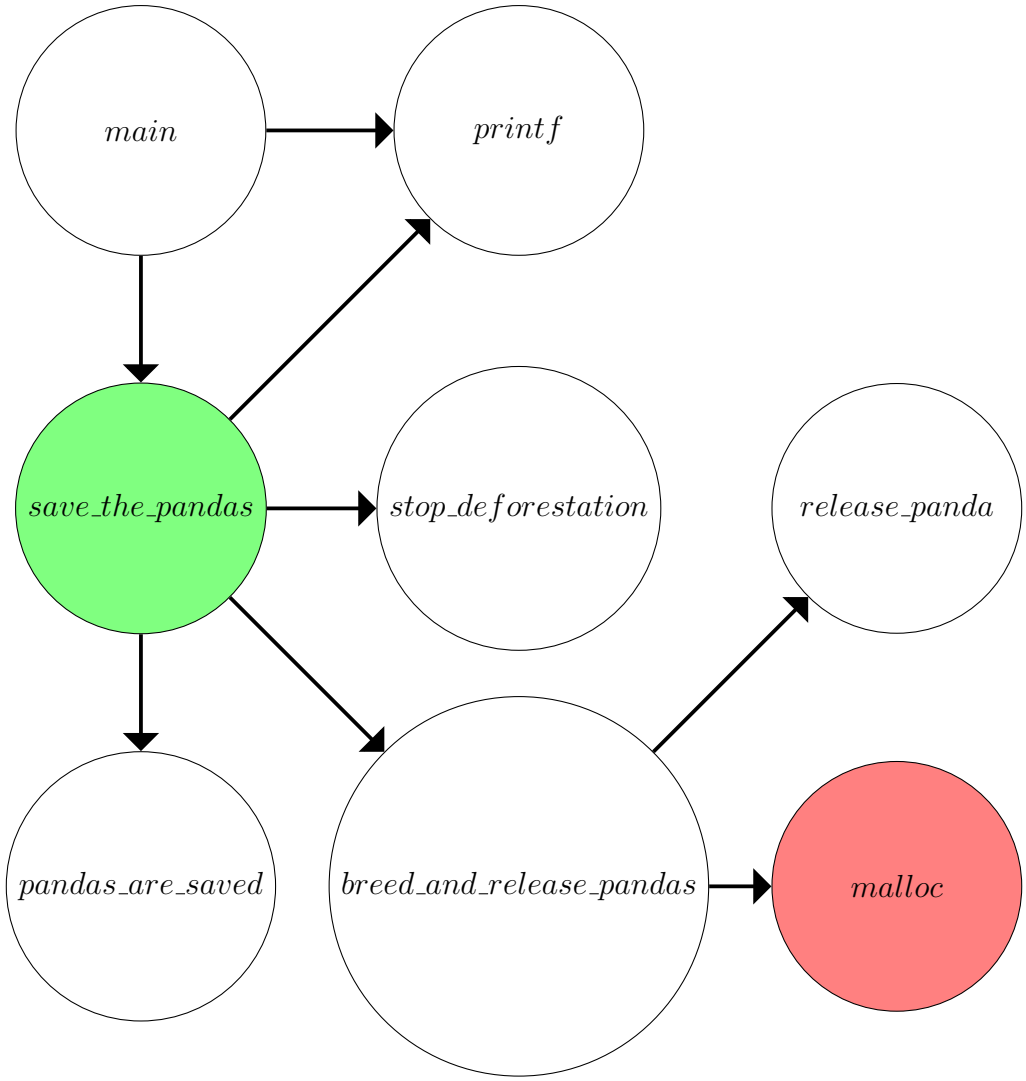

Figure 4.1: Color-coded Call Graph for Listing 2.1. Functions tagged `static_memory` are highlighted green and functions tagged `dynamic_memory` are highlighted red.

4.2. Function Pointers Pointing

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 explains the rules surrounding function pointers and explains how funqual tracks them in the call tree.

As a concrete example, refer to Listing 4.2. In this example, it is literally impossible to know what 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.

```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();
}
```

Listing 4.2: 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.

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 possile 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 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 but can be inferred statically for functions.

Listing 4.3 shows the same code as Listing 4.2 but with function types annotated. Figure 4.2 shows the call tree for Listing 4.3 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.

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

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

Listing 4.3: Same example program as Listing 4.2 but with function pointer type annotations inserted

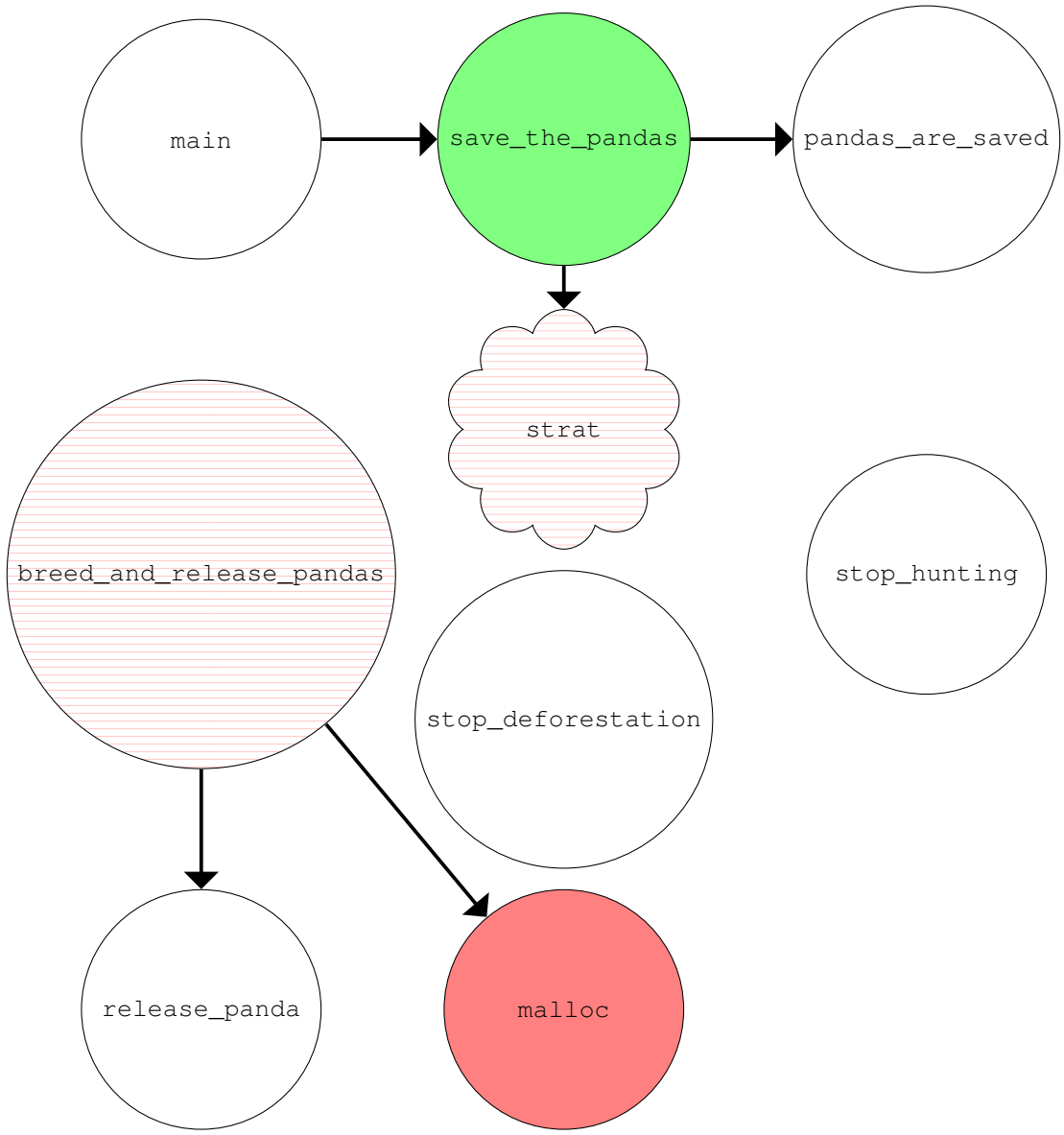

Figure 4.2: Color-coded Call Graph for Listing 4.3. Functions tagged `static_memory` are highlighted green and functions tagged `dynamic_memory` are highlighted red. Indirect types are represented as horizontal line patterns on a node. Clouds represent function pointers.

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 4.2 as red horizontal lines) is how we keep track of this possibility.

4.2.1. Indirect Type Inference

As mentioned earlier, funqual is able to infer the indirect type of functions. For any function, F, the indirect type can be determined by traversing the call graph and visiting all the vertices that are reachable from F. The indirect type of F is the union of the direct types and indirect types of all the vertices that F can reach.

Once again, the indirect type of F is a list of types that could *possibly* be reached from a call to F. If we are trying to enforce a $restrict\_indirect\_call(X, Y)$ rule, it is important that under no circumstance, following no execution path, is it possible for a function in X to call a function in Y. Taking this expansive view of indirect type is the only way to guarantee call graph safety.

This indirect type inference is only possible because the body of the function can be found and traversed statically. For function pointers, there is no way of knowing which functions the pointer can reference. As a result, indirect type for function pointers must be annotated explicitly and every assignment into the function pointer is checked to ensure that the assignment does not lose any type information.

4.2.2. Rules of Assignment

To properly enforce call tree constraints, funqual checks function pointer use 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 tree 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 the 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.

The general reasoning here is that in order to correctly enforce *require_direct_call* rules and *restrict_direct_call* rules, we need to know the exact direct type of all function pointers. In order to properly enforce *restrict_indirect_call*, we need to know all of the funqual types that are possibly reachable by a given function.

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

4.3. Call Graph Rules

Each Subsection here describes one of the call graph constratins supported by funqual. For each constraint, we explain the meaning of the rule, provide an algorithm that could check a call graph for violations, and present an argument for the algorithm's correctness with respect to the rest of the type system. The algorithms in this section are not the same as the algorithms actually implemented in the funqual tool, but instead are tools for argument.

4.3.1. Restrict Direct Call

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

A restrict direct call rule creates a constraint that functions with direct type

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 collumns 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.

X cannot call 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 and so is supported by funqual because it was easy to do so.

Listing 4.4 shows pseudocode for an algorithm that could 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.

```
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
```

Listing 4.4: Pseudocode for an algorithm that could 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.

This algorithm is relatively straightforward and runs in linear time with respect to the number of edges in the call graph. To defend the correctness of this algorithm we will put each function call in this graph into one of two possible cases: 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 have annotated the direct type of both the caller and the callee and we will assume these user-provided annotations to be correct. 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 an edge calling P will appear in `edges` and that edge will be checked in the same way as a standard function call.

4.3.2. Restrict Indirect Call

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

A restrict indirect call rule creates a constraint that functions in X cannot eventually lead to a call to a function in Y. This is a stronger constraint than the the restrict direct call rule because this rule restricts both direct and indirect calls. The need to enforce indirect calls in the presence of function pointers requires us to examine both the direct type of the callee as well as the indirect type.

Listing 4.5 shows pseudocode for an algorithm that could 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.

```
1 function enforce_restrict_indirect_call(X, Y, edges):
2     for edge in edges:
3         callee = edge.to
4         caller = edge.from
5
6         if callee is a regular function:
7             callee.indirect_type = infer_indirect_type(callee, edges)
8
```

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

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
```

Listing 4.5: Pseudocode for an algorithm that could 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.

The need to infer the indirect types of functions makes this algorithm much more complicated than the previous. As it is written, the complexity is $O(|E|^2 * |V|)$ in the

worse case but this algorithm could easily be improved by caching type inferences. Showing the correctness of this code is much more complicated but again can be simplified by breaking it down case by case.

In the event that there is a direct call from a function with direct type X to a function with direct type Y, this algorithm will detect it. Direct types are annotated statically so we can trust that `callee.direct_type` is correct. If a function with direct type X calls a function with direct type Y, then `edges` will contain that edge and the algorithm checks this case on line 9.

In the event that there is a direct call from a function with direct type X to a function pointer with indirect type Y, this algorithm will detect it. Indirect types for function pointers are annotated by the user and checked for each assignment and so we can trust that `callee.indirect_type` is correct. If a function with direct type X calls a function pointer with indirect type Y, then `edges` will contain that edge and the algorithm checks this case on line 11.

In the event that there is an indirect call from a function with direct type X to a function with direct type Y, this algorithm will also detect it. For this example, let function A call function B which calls function C. Also let A have direct type X and let C have direct type Y. When the loop in `enforce_restrict_indirect_call` reaches the edge from A to B, it will call `infer_indirect_type` to determine the indirect type of B. In doing so, the algorithm will visit the edge from B to C and will return an indirect type containing Y. The algorithm will see that A has direct type X and that B has indirect type Y and will terminate having detected the error on line 11.

In the event that there is an indirect call from a function with direct type X to a function pointer with indirect type Y, a similar sequence of events will occur. For this example, let function A call function B which calls function pointer C. Also let A have

direct type X and let C have indirect type Y. When `enforce_restrict_indirect_call` visits the call from A to B, it will infer the indirect type of B. During this process, it will visit all vertices reachable from B which includes C and it will return the union of all direct and indirect types visited which includes Y. As a result, Y is in the indirect type of B and the algorithm will terminate on line 11 having detected the error.

Lastly we need to argue for the assignment rules for function pointers. Let F be some function and C be some function pointer. Recall that for an assignment of F into C to typecheck, the direct types of F and C need to match exactly and the indirect type of C needs to be a superset of the indirect type of F. In this case, if at any point the user tried to assign the value of F into C and F had indirect type Y, then C would also have to have indirect type Y otherwise funqual would have reported an assignment type error. Assigning F into C necessitates that the indirect type of F be present in the indirect type of C. In the example where A calls B which calls C, we can be certain that when inferring the indirect type of B that it will visit C which must have an indirect type which accumulates the indirect types of all the things it could point to.

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.

Listing 4.6 shows pseudocode for an algorithm that could 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.

```
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 not in callee.direct_type:
            return false
    return true
```

Listing 4.6: Pseudocode for an algorithm that could 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.

To defend the correctness of this algorithm, we will put every function call into one of two possible cases: 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 trust the user in these regards. 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 any violations.

In the case of a function pointer call, we need to also examine all the possible assignments of 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.

4.4. Special Considerations when Creating a Call Graph

4.4.1. Dealing with Inheritance

According to the Liskov Substution Principal, "if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program". In this work, we assume this to be a basic principal of object oriented design. For the purpose of this paper, "if S is a subtype of T and M is a method of S, then calls to $T.M$ in a program may be replaced with objects of $S.M$ without altering any of the desirable properties of that program". As a result of this, when typechecking a call to $T.M$, we must also typecheck a call to $S.M$ to ensure that substituting S for M does not violate our call tree constraints.

In practice, this mean that for any method call from C to $T.M$ where C is the calling context and M is a method of T, we must add an edge in our call graph from C to $T.M$ and also from C to $S.M$ for any S that is a direct or indirect subtype of T.

Listing 4.7 demonstrates this concept. It is a piece of C source code that calls a virtual function. Figure 4.3 shows the call tree for this code sample.

```
class Panda {
protected:
    int m_hunger;
public:
    virtual int Feed() {
        m_hunger--;
    }
};

class RedPanda : public Panda{
public:
    int Feed() override {
        Stomach *stomach = malloc(sizeof(Stomach));
        memset(stomach, 0xBEEF, sizeof(Stomach));
    }
};

void feedPanda(Panda *panda) static_memory {
    panda->Feed();
}

int main(void) {
    feedPanda(new RedPanda());
}
```

Listing 4.7: Example C program demonstrating inheritance. In `feedPanda`, it is impossible to know statically which instance of the `Feed` function will be called. Figure 4.3 shows the call graph for this program.

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

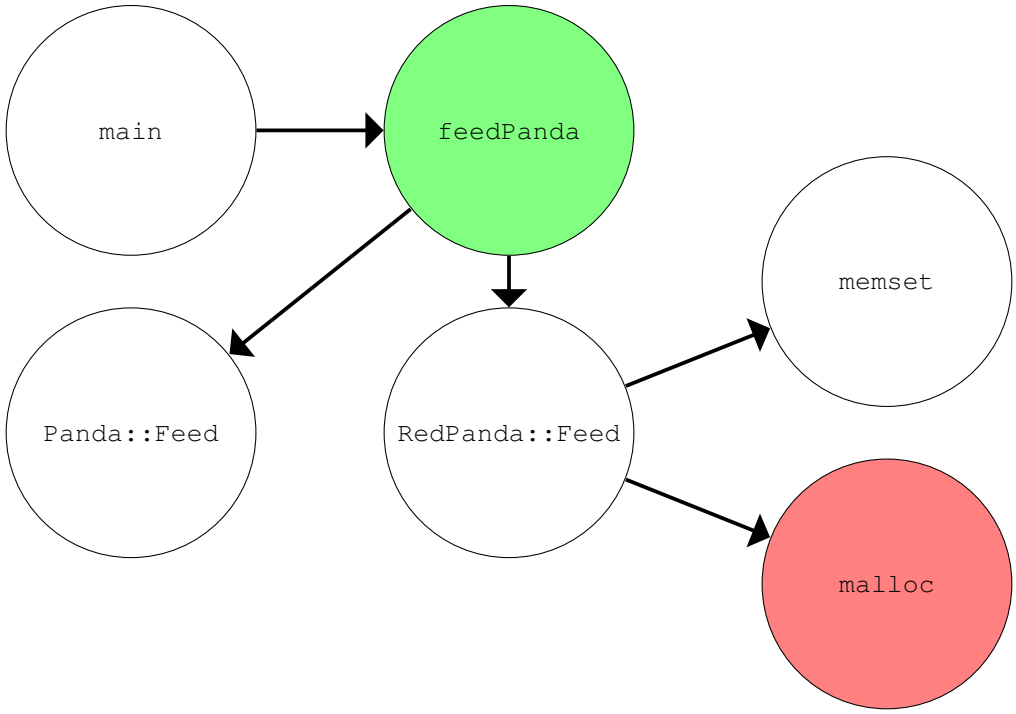

Figure 4.3: Call graph for Listing 4.7. Because `Panda::Feed` is a virtual function, we must draw an edge from `feedPanda` to every instance of `Feed`.

that any time `feedPanda` is called, it is impossible to know whether it is `Panda::Feed` being called or whether it is actually `RedPanda::Reed` being called. The only safe way to handle this scenario is to assume that `feedPanda` calls both of them. This is reflected in Figure 4.3 which is a call tree showing `feedPanda` pointing to both versions of the `Feed` function.

4.4.2. Operator Overloading

C++ allows for operator overloading. As a result, an expression such as `int 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.

4.4.3. Bridging the Divide between Translation Units

The compilation of C++ code is driven by translation units. Translation units are the files which are inputted into 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 not their value or definition (this is the case with extern variables, function prototypes, and class forward declarations). In these cases, examining the call tree of a single translation unit is not sufficient to enforcing global call-tree 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 tree and build a call tree which represents the entire codebase. In order to test this, we create several test cases where functions are defined in multiple translation units and where function a call tree 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

As mentioned earlier, 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, rule files, and the output that funqual will give.

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 syntaxes 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 macro is in scope. 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
#define QTAG(TAG) __attribute__((annotate("funqual::" #TAG)))
#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 effect 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 inline 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`.

```
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 rule files. By convention, rule files have the file extention `.qtag` but this convention 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 existance 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 proceeded by `-t` or `--tags-file` will be interpreted as a rules file. All other arguments will be interpreted as a translation unit. 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 Listing 4.7. 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.

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

5.2. Practical Limitations

Because of complexities in parsing C++, certain applications of function pointers are not checkable by funqual. Specifically, any expression where the lvalue in a function pointer assignment is anything other than a raw variable is not supported. Listing 5.1 shows a few examples of assignment expressions that funqual cannot parse correctly.

```
void *(**array)(size_t size);

array[0] = malloc; //assignment not checkable

struct {
    void *(*field)(size_t size);
} structure;

struct structure s;

s.field = malloc; //assignment not checkable
```

Listing 5.1: Examples of function pointer assignment expressions that are not parsed correctly by funqual

Additionally, if the source code being checked contains any syntax errors, funqual will fail to check the program and will print one of the syntax errors in the program.

This functionality is provided by libClang which we use to parse program source.

Chapter 6

APPLICATION

6.1. Glibc Nonreentrant Functions

The following functions are documented in glibc to be non-reentrant:

1. malloc
2. free
3. printf

There are others. We can mark interrupt handlers as one class and these functions as another and statially check that.

6.2. Restricting API available during initialization

My OS project was annotated so I didn't accidentally call malloc or printf before those things were initialized

6.3. Detecting noisy calls in high frequency contexts

Robotics was annotated and checked so I didn't accidentally have printf's in high frequency functions.

Chapter 7

FUTURE WORK

The following things are not handled correctly by the tool but it would be cool if they were:

1. struct fields
2. casting of function types
3. arrays of function pointers

future work should implement these features.

Chapter 8

CONCLUSION

It worked good. How do I measure that though?

BIBLIOGRAPHY

[1] Static analysis, November 2017.

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

[3] 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.

[4] J. Foster, M. Faehnlich, and A. Aiken. A theory of type qualifiers. May 1999.

[5] T. Glek. Dehydra, prcheck, squash in mercurial, July 2007.

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

[7] D. Greenfieldboyce and J. Foster. Type qualifier inference for java. 2007.

[8] Y. Kreinin. Defective c++, November 2016.

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

[10] H. Ogasawara, M. Aizawa, and A. Yamada. Experiences with program static analysis. 1998.

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

[12] 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.

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

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

[15] 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 in Software Engineering*, 32.