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

Table Page

LIST OF FIGURES

Figure Page

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 [7]. 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[7], some tools examine ownership of objects to enforce memory safety[5], some tools examine the flow of values through a program to ensure security[4], 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 section aims to provide context for the work done in this paper as well as provide some intuition behind funqual works the way that it does. The first section here touches on the kind of type system which should be familiar to most programmers. The second section here develops the concept of a call-graph and demonstrates how a type system might operate on it.

Type Qualifiers on Variables

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 [1]. 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 [1]. 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 if it a $constint$ or a $constchar*$ or a $constPanda$ - 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, the compiler writers, or the super-user. 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.

Type Qualifiers on the Call Graph

The focus of this paper is on creating and assigning 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.

The call graph of a program is a directed graph where every function is a node and where function calls are edges directed from the caller to the callee. The type qualifiers in this context are applied to the edges and the things we wish to constrain are connections between edges. Below is an example of a C program as well as the associated call graph.

```
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");
  }
}
```

As demonstrated by figure 2.1, 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. We can say that *main* directly calls *printf* and *save_the_pandas* and that *save_the_pandas* directly calls *pandas_are_saved*, *stop_deforestation*, and *breed_and_release_pandas* because there is an edge in the graph that directly con-

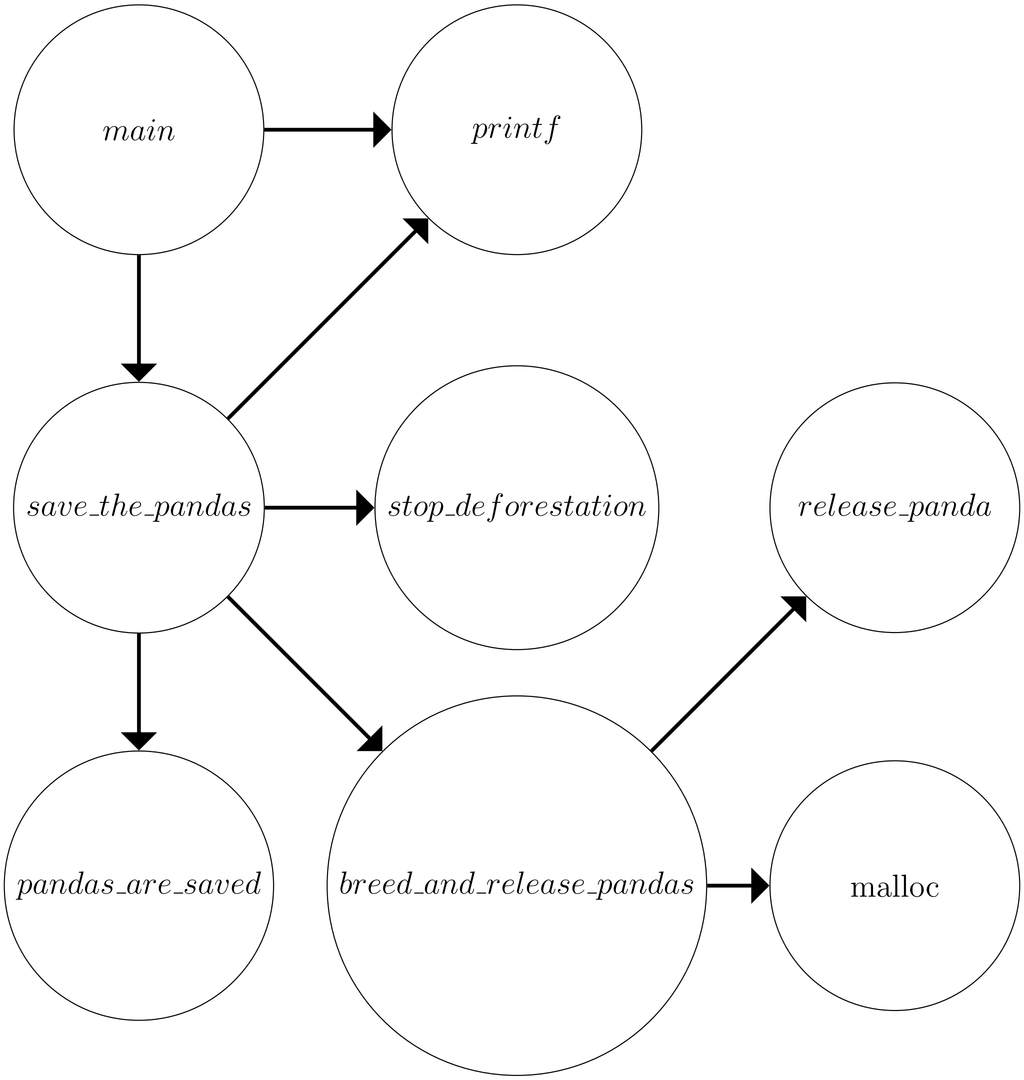

Figure 2.1: Call Graph for the Save the Pandas code listing

nects these functions. We can also say that *main* indirectly calls *stop_deforestation* because there is a path from *main* to to *stop_deforestation*.

Let us now imagine that there is some constraint whereby *save_the_pandas* is not allowed to touch the heap. Using existing tools, it would be possible to bar any function anywhere in the codebase from calling *malloc* or to simply link to a nonstandard library without *malloc* defined. This solution is problematic, however, because now no entity in the entire codebase can call *malloc.* What would be more useful is a system of marking functions that cannot call *malloc* and having a tool check the call graph to make sure it does not happen.

Chapter 3

RELATED WORK

The past few decades have seen a huge surge of research into type systems. Where much of the original research has been in making type systems which make it easier for the compiler to produce efficient machine code, recent research has focused on making type systems which are intuitive and helpful to the human code author. Much of this research focuses on refining the types of variables used in expressions. This paper instead focuses on the types of functions and the context from which they are called. This section explores some of the expression-based type system refinements and contextualizes them with respect to this research.

jQual

jQual is a research project aimed at providing a system of user-defined type qualifiers to the java programming language. The intent is to allow the user to define their own qualifiers that can refine types and that can be checked statically [4, 3]. Much of the focus on this work is in type inference. All type qualifiers are constraints on the types of constants and variables. jQual has no concept of a function type qualifier other than the qualifiers of parameters and return types.

Related to the jQual project, cQual is a project aimed at providing a system of user-defined type qualifiers to the C programming language. The initial contribution was a program that could analyze program source and determine where additional consts may fit [1]. Much of the theoretical background for subtyping and supertyping in this paper comes directly from this work. However, no reference is made to the possible typing of functions.

Chapter 4

TYPE RULES

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. The first section simply explains the process of marking functions. The second section shows what types of rules are supported by funqual. The third section demonstrates special considerations made for function pointers and explains the rules for their use. The final section explores the universe of special considerations and compromises made when creating the call-graph to be checked.

Note this section focuses only on the conceptual design of funqual. For any details on how to actually use it, refer to to 5.1.

Overview

Function Qualifier Annotations with QTAG

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 checkable by funqual with no modification. To this end, we use the existing C++17 annotation syntax to mark code.

For clarity and convenience we assume the following macro is in scope. In practice, this macro can be repeated in the codebase or included in files containing function annotations:

```
#ifndef QTAG
#define QTAG(TAG) __attribute__((annotate(”funqual::” #TAG)))
#endif
```

Note that the __attribute__((annotate(foobar))) syntax is generally used for compiler-specific directives (like packed, allign(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. The following are a few examples:

```
void *malloc(size_t size) QTAG(dynamic_memory);
```

```

// multiple qualifiers can be added at once like so
int printf(const char *__restrict __format, ...) QTAG(io) QTAG(blocking);

// the same function can be annotated in multiple places like so
// funqual will enforce the union
void *malloc(size_t size) QTAG(blocking);
```

Basic Rules

Restrict Direct Call

$$\mathit{restrict\_direct}(X, Y) = (V \in X \implies Y \notin A) \mid (V, A) \in G$$

A restrict direct call requires that functions in set X only ever call functions in set Y. This constraint can be checked in time that is linear with the number of function calls in the program and can be reported very easily. An example use case for this type of restriction might be to restrict realtime functions to only calling other realtime functions.

Restrict Indirect Call

$$\mathit{restrict\_indirect}(X, Y) =$$

Require Direct Call

$$\mathit{require\_direct}(X, Y) = (V \in X \implies Y \in A) \mid (V, A) \in G$$

A require direct call requires that functions in set X never call functions in set Y.

Function Pointers Pointing

Basic Annotation and Direct Type with QTAG

Indirect Type with QTAG_IND

Rules of Assignment

Special Considerations when Creating a Call Graph

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 where the preprocessor has already expanded all macros (including *include* substitutions). During this process, many symbols are said to have *externallinkage* 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 be able to see which internally linked functions call externally linked functions but not vise versa.

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.

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 and build off it. 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.

Below is a code example that demonstrates this rule in use:

```
void *malloc(size_t size) FUNQUAL(dynamic_memory);

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

class RedPanda : public Panda{
public:
    int Feed() {
        char *buff = malloc(30);
        m_hunger--;
    }
};

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

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

This example shows a function called feedPanda which calls *Panda.Feed*. This function also shows that *malloc* is tagged with *dynamic_memory* and that *feedPanda* is tagged with *static_memory*. Presumably there is a indirect call restriction that prevents functions tagged with static_memory from calling (either directly or indirectly) functions tagged with dynamic_memory. It we simply looked at the type of *panda*, we would falsely believe that this program is typesafe. However, we see that it's possible for the actual type of $panda$ to be $RedPanda$ in which case we call $RedPanda :: Feed$ which calls $malloc$ which violates our static memory constraint. To solve this problem, we must create a call graph which contains edges pointing to both $Panda :: Feed$ and $RedPanda :: Feed$.

Function Pointers Pointing

Just like functions, function pointers need to have their place in the call graph. Function pointers are difficult because we can't always know at build-time what they refer to. With a standard function, we can simply build a mapping that goes from the function to its body. With function pointers, we have to be able to accommodate for the pointer referencing different functions at different times.

To solve this problem, we allude to the existing type system on function pointers. According to the C++ standard, in order to assign a function pointer to reference an existing function, the type of the function pointer must match the type of the function being referenced. We continue this trend by forcing the type qualifiers of the function pointer to match the type qualifiers of the function being referenced. These type qualifiers can, of course, be forcibly removed by a cast in extreme circumstances, but we believe that this rule ensure consistency between within the call tree in the face of function pointers.

Checking the Call Graph

This section contains explanation and motivation behind the call graph rules implemented in this tool.

Chapter 5

IMPLEMENTATION

Operation

Chapter 6

APPLICATION

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.

Restricting API available during initialization

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

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

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

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

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

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

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

[7] 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.