

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

© 2018  
Andrew Nelson  
ALL RIGHTS RESERVED

## 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: Aaron Keen, Ph.D.  
Professor of Computer Science

COMMITTEE MEMBER: Aaron Keen, Ph.D.  
Professor of Computer Science

## ABSTRACT

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

Andrew Nelson

In this paper we create a static analysis tool called funqual. Funqual 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. 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 errors in code before that code is even run.

## ACKNOWLEDGMENTS

Thanks to:

- Andrew Guenther, for uploading this template

## TABLE OF CONTENTS

	Page
LIST OF TABLES . . . . .	vii
LIST OF FIGURES . . . . .	viii
CHAPTER	
1 Introduction . . . . .	1
2 Background . . . . .	3
2.1 Type Qualifiers on Variables . . . . .	3
3 Related work . . . . .	4
3.1 jQual . . . . .	4
BIBLIOGRAPHY . . . . .	5
APPENDICES	

## 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. 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. All these tools have different use cases and can be used in conjunction to write code that is error-free.

Languages like C++ and java are well suited for static analysis because type information is readily and explicitly available in the source code. In fact, a lot of static analysis happens during the compilation phase. This analysis finds and reports bugs like type mismatches, undefined references, and conflicting definitions.

The following snippet of code demonstrates this concept:

```
int main(void) {  
    int i = 9;  
    char *j = "pandas";  
    return i + j;  
}
```

If C somehow did not perform any sort of static analysis, this would be a runtime error (or worse may fail silently at runtime). GCC, however, checks the types of all operation and operands at build-time in order to build efficient machine code. When compiled with GCC, the code above gives an error about addition between `int` and `char*`. While inexperienced programmers may find this knack for finding type-errors to be cumbersome, more experienced programmers often learn to love

this static checking that comes for free with the language.

Of course, there are things which are impossible to statically check in C and C++. For years, various projects have tried to build tools which can statically analyze code to check for memory errors, unit errors, and infinite loops. Unfortunately, many of these projects require specific language extensions in order for there to be enough information in the source code for the tools to work. This is unfortunate because it prevents the tools from being used on code "in the wild", or code that has not been written with the tool in mind.

In this paper, we create a tool called funqual which can statically check the call-tree of C++ code "in the wild" for certain patterns and report back to the user. The program uses libclang to parse C++17 code and build a call-graph for the entire program. The program then checks this call-graph against user-defined rules which encode.

## Chapter 2

### BACKGROUND

#### Type Qualifiers on Variables

In traditional research into type-systems, Type Qualifiers are a way to refine 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 typed 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].

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 refining 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 em

## 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 [3, 2]. 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.

## BIBLIOGRAPHY

- [1] J. Foster, M. Faehnlich, and A. Aiken. A theory of type qualifiers. May 1999.
- [2] D. Greenfieldboyce and J. Foster. Type qualifiers for java. August 2005.
- [3] D. Greenfieldboyce and J. Foster. Type qualifier inference for java. 2007.