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:	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 rollows 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:

- 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

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 [4]. 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:

```
1 int main(void) {
2   int i = 9;
3   char *j = "pandas";
4   return i + j;
5 }
```

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 in 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. A classic study by the univeristy of North Carolina in conjunction with Nortel Networks found that the use of automated static analysis can detect certain types of programming errors at approximately the same accuracy as manual code inspection and that this checking can be performed in a fraction of the time [4].

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 wile" 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

This section aims to provide context for the work done in this paper as well as provide some intuition behind ow the solution here was reached. The first section here touches on the kind of type system which should be familiar to most programmers. The second section here demonstrates a different sort of type system which may not seem as familiar to readers. A general explanation of the call tree is given in this section, though it will be developed in more specific detail later in the paper.

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 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 emphasis on creating project-specific qualifiers to help maintain program semantics.

Type Qualifiers on the Call Tree

The focus of this paper is on assigning and qualifying types of functions and enforcing constraints on where they can and cannot be called. The central notion behind this sort of type checking is that every program has a call tree and that there are certain patterns in the call tree which must be prevented.

The call tree 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 tree. The notion of a call tree will be expanded on later in this paper but the general concept is demonstrated here.

```
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.

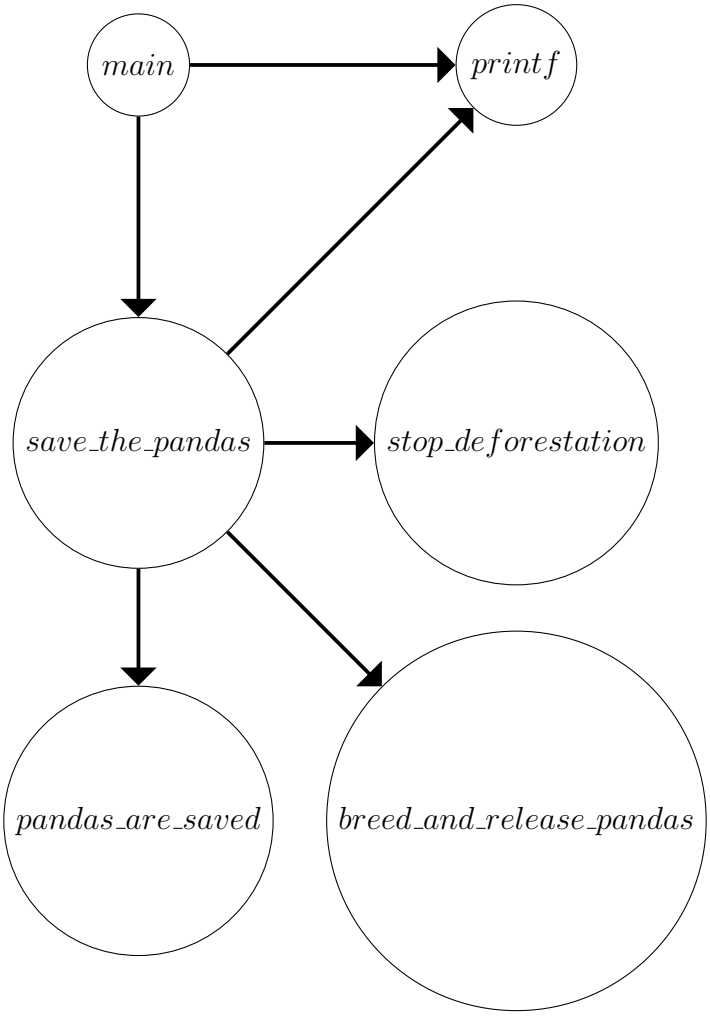

Figure 2.1: Call Graph for the Save the Pandas code listing

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.

Chapter 4

VALIDATION

This chapter examines some of the unique properties that a tool like funqual needs to respect in order to function properly as well as how those properties were tested both in the code and in practice.

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:

```
1  void *malloc(size_t size) FUNQUAL(dynamic_memory);
2
3  class Panda {
4  protected:
5      int m_hunger;
6  public:
7      int Feed() {
8          m_hunger--;
9      }
10 };
11
12 class RedPanda : public Panda{
13 public:
14     int Feed() {
15         char *buff = malloc(30);
16         m_hunger--;
17     }
18 };
19
20 void feedPanda(Panda *panda) FUNQUAL(static_memory) {
21     panda->Feed();
22 }
23
24 int main(void) {
25     feedPanda(new RedPanda());
26 }
```

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.

Restrict Direct Call

$$restrict(X, Y) = (V \in X \implies Y \in A) \mid (V, A) \in G$$

A restrict direct call requires that functions in set X only ever call function sin 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

$$restrict(X, Y) =$$

Call Tree Length

In cirtain circumstances (especially in embedded and kernel programming), the stack size is very limited. It becomes important to statically ensure that the stack size of the program does not exceed a certain number of frames. To do this, our tool can examine certain functions that are tagged with a stack size restriction and check all out-paths in the codebase to ensure that this function's stack size never exceeds the given number of frames. This is modeled as a depth first search that bottoms out at a given depth and reports a warning to the user.

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.

[4] 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.