Program Analysis for Introductory Education: Leveraging Programmer Specifications

Jason R. Koenig

August 2014

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Chair
André Platzer

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This research was supported by a scholarship from the Siebel Scholars Foundation and generous support from the Microsoft Corporation.

Abstract

Students in the second introductory programming course at Carnegie Mellon routinely have trouble learning how to properly use pointers. The course is taught in C0, a simple, safe C-subset. A common error in student's code is to dereference a pointer which may be null. The goal of this thesis is to evaluate whether automated program analysis can provide useful feedback to students. Typically program analysis is difficult even for experts to operate, because the analysis must consider the whole program at once. Instead of these approaches, we use student supplied specifications, such as pre- and postconditions, to guide the analysis. We developed a tool which analyzes a program and determines whether each dereference in a program is guaranteed to be safe, will definitely cause a crash, or is unknown. We ran our tool on a corpus of student homework submissions from several semesters of the introductory programming course, and evaluated a random subset of the errors to determine whether they were accurate. We determined that on simple assignments, a large fraction of the errors were accurate, but that on more complicated assignments with complex invariants, the rate of undesirable errors was much higher. Future extensions of this work would compare student learning with and without the tool, as well as investigate ways to improve accuracy on the more involved assignments.

Contents

Chapter 1

Introduction

When the faculty of Carnegie Mellon reworked their core computer science curriculum, they had a chance to redesign the introductory courses for incoming undergraduate freshman. That design process resulted in a new course for teaching programming, data structures, and algorithms, along with a new programming language to accompany it. The idea of this course was to teach basic algorithms and data structures while emphasizing a principled way of programming. The language, `C0`, was designed to facilitate this: eschewing esoteric corner cases and embracing constructs which are easy to reason about. The language had to be both a platform on which students could code their first linked list and hash tables, as well as prepare them for the next course taught in full `C`.

The key idea to enable all of these course goals in a single semester was to embed programmer supplied specifications into the language. By having the programmer write down explicitly the various conditions and invariants, the language could (1) help students catch errors earlier by dynamically checking that they are satisfied and (2) help them understand the operation of complex algorithms. In addition to being helpful for understanding an algorithm, these invariants allow one to reason rigorously about the operation of the program.

By teaching students not just to write code, but also specifications for its operation, the course introduces students to the ideas of separation of interface and implementation, reasoning about correctness, and computational thinking. Specifications allow the program to be decomposed using *contracts*, where one part is responsible for establishing a condition and another part relies on this condition. They also form the basis of formally reasoning about the operation of the program. These benefits depend on students actually writing specifications, however. Over the semesters, a particular pattern arose for students working on their programming assignments. First, students relied on the automated test suite available on the submission server to determine the correctness of their programs, which led to long pauses before they would receive feedback. Second, while students would write some specifications when the assignment would specifically test them, they would avoid adding specifications to their own code.

In this thesis, we aim to tackle both problems by introducing the capability for the C0 compiler to give errors for potential runtime errors. Like any good compiler, C0 gives errors for syntax problems, type errors, etc.:

```
error:expected ')', found: '='
[Hint: assignment l = e not permitted as expression;
use '==' for comparison?]
 if (a = NULL)
       ~
```

Our goal is to have the same easy feedback for higher level semantic errors, including missing specifications and operations which may crash the program at runtime:

```
dict.c0:82.12-82.22:error:unprotected pointer dereference
[hint: list may be null]
    while (list->next != NULL)
           ~~~~~~~~~~
```

We want to give students fast feedback which they can use to check their understanding, catch errors before running a test suite, and encourage students to write more specifications. Because of these goals, we created an analysis which reasons about the code much in the same way a student might, looking locally at each functions rather than trying to understand the whole program. Our analysis is relatively simple compared to most static analysis tools, because it is designed to be used and understood by freshman.

Complex specifications can be difficult for the analysis to understand, but also difficult for a human to understand as well. The correctness of a C0 program can mean many things: the program does not crash on any input, the program is well specified, the program achieves the goal, the program operates efficiently, etc. Even if a program does not crash, if it is not easy to see why the program is safe, we may say that it is still incorrect at some level. Thus even if a more powerful analysis is able to certify that the program is safe, the limitations of a simple analysis may be beneficial in encouraging students to make their programs more direct.

1.1 Overview

This thesis is split into three primary sections. The first, Chapter 2, gives a brief introduction to C0, including the language itself and the larger context for which it was designed. The second, Chapter 3, details the framework we have developed for analyzing C0, including how we handle the reasoning locally and constructs which typically frustrate static analysis. The third, Chapters 4 and 5, presents the specific task of capturing the behavior of pointers and the heap, and present the results of running this analysis on the corpus of student submissions.

Chapter 2

C0 Overview

C0 is a educational programming language designed as in introduction to C. C0 is an subset of C which removes non-type safe and undefined behavior, and adds support for programmer supplied specifications such as pre- and postconditions. The language has a simple, well specified semantics, including a definite execution order, strict type safety, and a garbage collector. The language is explicitly designed for teaching first and second semester students who only have a semester of programming experience. More information on C0, including a tutorial, may be found at [2].

2.1 Design Goals

Because C0 is a language designed explicitly for a specific educational context, it has a different set of goals from a general purpose programming language [1]. These goals inform the features and design decisions that were made in the construction of the language. Further, these educational goals apply to our contribution as well: we are building a tool with the purpose of helping students learn. Because we are not, for example, verifying industrial scale software, our tool differs from typical program analysis tools. To understand how our contribution fits into this larger context, we need to start with the goals for the course and curriculum that both C0 and our tool are designed to support.

The C0 language is designed as the teaching language for Principles of Imperative Programming (15-122), the second introductory programming course at Carnegie Mellon. Because the next course in the curriculum make heavy use of C in teaching systems concepts such as stack and heap layout, assembly programming, process management, and UNIX fundamentals, it is important that C0 prepare students for using C. In fact, the final few weeks of 15-122 are taught in full C, so the transition must be quick. On the other end, because it is the second course in the curriculum, students in 15-122 have only one semester of programming experience. Thus the language is designed with a limited scope and easily explainable semantics. Because poorly written C can be especially difficult to debug and maintain, the course is designed to teach students a good idiomatic style which avoids traditionally error prone `C` constructs.

In addition to teaching students how to construct programs, the course teaches students how to reason about programs. This reasoning includes skills such as reading a program and determining what it accomplishes, constructing an argument or informal proof for why a program correctly implements a task, and understanding how a program executes on concrete or abstract inputs. The goal is thus to teach students not just to write programs, but also to have confidence that the code they wrote does what they intended. To support this, the course teaches students to write pre- and postconditions, loop invariants, and assertions, in addition to the program code. These specifications have two effects: they help students isolate bugs and they enable local reasoning about programs. Because specifications are part of the program, they can be checked dynamically and cause bugs to fail as close to point of introduction as possible. In addition, the program can be understood part by part, because the specifications break the program at function and loop boundaries. A core idea of the course is that these specifications reduce the complexity of teaching program reasoning, so that it can be taught in addition to programming skills, data structures, and algorithms in one semester.

2.2 Design Decisions

The goals of the language are thus to prepare students for programming idiomatic `C`, with an easily explainable semantics, while encouraging them to write down specifications about their program. Because students will need to transition to `C`, `C0` is a almost completely a subset of `C`. It has a much stricter syntax, where for example assignment cannot accidentally be substituted for equality comparison. Operations which are meaningless, such as accessing an array out of bounds or dividing by zero, are guaranteed to abort the program rather than causing insidious effects later. Core elements of `C`, such as integers, functions, most of the control flow statements, bitwise operators, arrays, structs, and pointers are retained. The more complicated or error prone elements are omitted, such as unions, pointer casting, floats, undefined and implementation defined behavior, linking and the build process, static and global variables, and the preprocessor. Additions to `C` are a garbage collector, complete type safety, a simple `#include` replacement, and the specifications mentioned previously.

Because it is not a general purpose programming language, many features from `C` can be omitted if they are not necessary for programming basic algorithms and data structures. The limited scope of `C0` means that the entire language can be taught in a semester. The language is designed so that students can reason about the behavior of their programs at the level of `C0` source code, without having to resort to a translation to a low-level machine model as is typical with `C`. For example, strict type safety means that a field in a struct can only change if there is a syntactic assignment to that field in the program. Variables on the stack cannot be changed by out of bounds writes to adjacent arrays, as is possible in `C`. Operations have defined semantics (aborting the program is well defined behavior) in all cases, rather than silently corrupting memory in unpredictable ways. By omitting `free()` and using a garbage collector, type safety can be ensured, and students do not need to learn manual memory management.

One decision particularly relevant to our contribution is that the specifications are written as expressions in the `C0` language rather than a separate logical language. This means that students only have one language to learn rather than two, and helps to keep the scope of the language small. However, this means the analysis must extract information from imperative code rather than a well behaved logic based language.

2.3 Semantics

The semantics of `C0` are specified via a syntax, a static semantics, and a dynamic semantics. We will not give a full formal semantics here, as this is outside the scope of this work and is presented in [7]. Further, we will elide elements such as the library system for interfacing with `C` code that are not relevant to our contributions.[1] Instead we will explain the features of `C0` that affect our work and how `C0` differs from `C`.

2.3.1 Syntax

A `C0` program is composed of a series of declarations and definitions of structures, functions, and typedefs. As in `C`, functions and structures can be declared before they are defined, and typedefs can be used to give an alias to the name of a type. The `#use` declaration is roughly similar to the `#include` directive from `C`, except files are loaded at most once without the need for the idiomatic preprocessor guards. No other `C` preprocessor directives are available. There are no global variables, so the only state a function can access is that which is reachable from its arguments.

Functions contain statements, such as `if`, `while`, and `return`. Each function has a signature, giving the types of its arguments and the return value (if it is does not return `void`). The statements have their usual structure from `C`, except where specifications are involved. Expressions are also like their `C` counterparts, except with more restrictions on valid forms. For example, equality (`==`) is not defined for string types; they must be compared with the library function `string_equal`.[2] The assignment and increment statements, *lv* `=` *e* and *lv*`++` are statements, never expressions, which prevents a whole class of common errors in `C`. The derived form *e*`->`*f* is a common abbreviation for `(*`*e*`).`*f*. A subset of the syntax of `C0` which contains the portions relevant to our contribution is given in Figure 2.1. We omit the details of lexing and parsing, such as disambiguating

[1]These functions are handled by using their specifications, and observing that a vast majority cannot visibly modify memory because user structs are not reachable from their arguments.

[2]This is because in `C`, comparing strings using `==` compares their addresses, not their values.

the context sensitive grammar and operator precedence, because these elements are inherited from C and do not affect our analysis.

2.3.2 Types

The type system of C0 is very simple. There are a few built in types: `int`, `string`, `char`, and `bool`. Structs are types as well, with the syntax `struct s` for a struct with name `s`. `typedef`s allow aliases for an existing type to be made, but `typedef`s cannot themselves introduce a recursive type and are thus syntactic only. The type forming operators `*` and `[]` may be applied to a type t to form pointers to cells of t and arrays of t respectively. Usually pointers point to struct types, and idiomatically we typedef the name `s` to the type `struct s*` (structs and types have their own namespaces). For the purposes of this contribution, all types can be expanded to some sequence applications of `*` and `[]` to a built-in type or a struct, effectively expanding typedef definitions.

All types are *small* except struct types, which are *large*. Only small types may be passed as arguments or stored in variables. These restrictions mean all struct type expressions must be immediately surrounded by a field access. Large types are permitted as fields and array elements however, so a struct or array can contain structs without pointer indirection.

One important property of C0 is strict type safety. Because all pointer types are distinct and there is no casting operator, pointers to a given type may only be obtained from allocating that type. Because this allocation guarantees the memory does not overlap with any other allocated memory, we have the property that fields, array elements, variables on the stack, etc. may only be modified by a syntactic assignment to that value. This is a important property for our analysis, and also for student understanding.

Functions may return `void`, which indicates a lack of return value. `void` is not allowed anywhere else, however, which rules out `void*`.[3]

2.3.3 Dynamic Semantics

The dynamics of C0 are very similar to C. The execution of the program begins with `main()`, and continues by executing functions. The complete state of the program is the function stack, the executing function, and a heap with allocated arrays and structs. Because there are no global variables, state is not accessible to functions except through their arguments, either directly or through pointers. Functions are not first class, so the function at each call site is fixed.

Execution within a function occurs via the familiar statements `if`, `while`, `return`, etc. Unlike C, C0 has a defined execution order for every statement and expression, which is usually left to right. The meaning of each statement is as follows:

[3]A work in progress language superset known as C1 will provide a type safe way to use `void*` to create generic programs.

```
dfn  ::=  struct structname { τ1 f1;τ2 f2; ... }
      |   τ g(τ1 v1, ..., τn vn)
            //@requires e;
            //@ensures e;
          { s }
      |   typedef τ typename;

s    ::=  if (e) s1 else s2
      |   while(e) //@loop_invariant e; s
      |   for(s; e; s) //@loop_invariant e; s
      |   return e; | return;
      |   { s s ... }
      |   assert(e);
      |   error(e);
      |   //@assert e;
      |   lv = e; | lv op= e;
      |   τ v = e; | τ v;
      |   lv ++; | lv --;
      |   e;

lv   ::=  v | lv.f | lv->f | *lv | lv[e] | (lv)

e    ::=  num | 'c' | "str" | true | false | NULL
      |   v | e op e | op e
      |   e ? e : e
      |   g(e1, ..., en)
      |   e.f | e->f | e[e] | *e
      |   alloc(τ) | alloc_array(τ, e)
      |   \length(e) | \result

op   ::=  + | - | * | / | % | ! | ^ | & | | | ~ | << | >>
      |   && | || | < | <= | == | != | >= | >
```

Figure 2.1: The abbreviated syntax of C0.

```
τ  ::=  int | string | char | bool
    |   τ*
    |   τ[]
    |   typename
    |   struct structname
```

Figure 2.2: Types in C0

1. `if` (e) s_1 `else` s_2. This statement evaluates e, and then executes s_1 or s_2 if e is true or false respectively. `else` s_2 is optional, in which case nothing will be executed if e is false.

2. `while(`e`)`
   `//@loop_invariant` I`;`
   `{`b`}`

   A while statement consists of a loop guard, possibly invariant(s) I, and a loop body b. First, the invariant I is checked. Then, the loop test is evaluated. If it is true, then the loop continues with the body. If it is false, then the loop exits and the next statement after the loop is executed. After the loop body, the invariant is checked again and the process resumes. Regardless, we know after the loop that the invariant holds, because it held initially, and it held after every execution of the loop body.

3. `for(`s_1`;`e`;`s_2`)`
   `//@loop_invariant` I`;`
   `{`b`}`

   A for loop may be expanded into a while loop by placing s_1 before the loop, using the invariants and the exit condition e, and putting s_2 after b in the body of the `while`. If a variable is declared in s_1, then it is bound in the invariants, e, s_2, and b.[4]

4. lv `=` e. This statement assigns a value computed by e to a location denoted by lv. Due to the left to right evaluation order, we first evaluate lv to determine the location to write to (which may fail because an array is accessed out of bounds, for example). Then we evaluate e, and finally perform the assignment. One special case is that if $lv = *p$ for some pointer p, we check whether p is `NULL` after evaluating both lv and e. Evaluation of p itself happens first, and may encounter errors if it involves dereferencing `NULL`. If lv is instead a field access or an array access, then array bounds and pointer dereferences are checked during the evaluation of lv.

5. lv op`=` e. Assignment with a binary operator is roughly equivalent to lv `=` lv op e, except that lv is only evaluated once. Because a value is needed, when lv is a dereference (`*`p), the pointer is checked for `NULL` along with the other effects of p, rather than after e is evaluated. If the operation op may fail, as in division by zero or shift by a large value, then this condition is checked after lv and e are evaluated, but before the assignment.

6. Expression. An expression can be a statement, in which case the return value, if any, is simply ignored. The effects of the expression, such as from function calls or unsafe operations, are still executed. Expressions

[4] s_1 and s_2 are optional, but e must be a non-empty expression. s_1 and s_2 must be variable assignments or declarations, `e++`, `e--`, or expressions statements.

includes `void`-type function calls, which can only be used as statements, as no expressions take a `void`-type subexpression.

7. `//@assert` *e* and `assert(`*e*`)`. An assertion aborts the program if the supplied expression is not true. This kind of failure is less serious than actually encountering a null pointer dereference or other error because in `C` the later would be undefined behavior, while the former always aborts. The specification form (`//@assert`) is only enabled with the `-d` flag for dynamic checking of specifications, while the second form is always checked.[5]

8. `error(`*m*`)`. The `error` command immediately aborts the program. This is used to signal problems in the input or environment, rather than internal bugs in the program, which should use specifications.

9. `lv++` and `lv--`. These are interpreted as `lv += 1` and `lv -= 1`, respectively.

Expression evaluation is more rigid than it is in `C`. Expressions can either evaluate to a value normally, or they can abort the program execution. In general an expression may have side effects on the heap, but not local variables. The order of evaluation is important to the semantics of `C0`, because it specifies which errors are raised and the order of effects on the heap. Most expressions evaluated from left to right. The simplest expressions are the standard binary and unary functions `==`, `!=`, `<<`, `>>`, `<`, `<=`, `>=`, `>`, `&`, `|`, `^`, `+`, `-`, `*`, `/`, `%`, `!`, `-` (unary), and `~`. These all evaluate their left argument, then their right argument, and then compute the result. Unary forms just evaluate their single argument.

For some expressions, such as division and shifts, if the values are out of range then the program will abort. Similarly, for dereferences `*`*e* and array access *a*`[`*i*`]`, the program will abort if the pointer is `NULL` or the array index is not within the length of the array.

The expressions `&&`, `||`, and the conditional `? :` may only evaluate some of their sub-expressions, i.e. these operators are *short-circuiting.* These always evaluate their leftmost argument. The conditional then evaluates one of the other two depending on whether the first was true or false. For logical "and" and logical "or", the second argument is evaluated only if the first does not determine the overall result. For `&&`, this means the second is evaluated only if the first is true, and for `||` the second is only evaluated when the first is false. This allows the programmer to write code like `p == NULL || p->f != NULL`, which regardless of the value of `p`, will never abort.

It is this abortion of the program through these erroneous expressions that we are concerned with preventing. In `C`, these errors may not necessarily trigger the program to crash; sometimes the program continues on in a corrupted state. Further, the compiler optimizer is allowed to assume that these errors never occur, thereby doing something the programmer did not expect. In `C0`, these errors always cause a deterministic program abortion, which is ensured by

[5]One can think of the specifications as present for "debug" builds, while assert is used for checks which should remain even in a "release" build.

```
int func(int arg)
//@requires P;
//@ensures Q1;
//@ensures Q2;
{
    while(cond)
    //@loop_invariant I;
    {
        //@assert A;
        ...
    }
    return e;
}
```

Figure 2.3: A simple function in C0, with all four specification forms.

the compiler by inserting dynamic checks for these conditions. However, when coding in C, these conditions are not checked, and it becomes imperative to avoid crashing the program or executing undefined behavior.

Pointers in C0 are safer than in C because the operations on them are very restricted. There is no "address of" operator (& is exclusively bitwise and), and no casting. There are only two ways to obtain a pointer: through the syntactic construct alloc(τ), whose type argument gives the type of the memory which will be allocated, or the constant NULL. alloc(τ) is roughly equivalent to (*τ)malloc(), except that if C0 runs out of memory then the program aborts. There is no corresponding free, because type safety allows garbage collection. This means that there is a language level invariant that pointers are always either NULL or point to a valid memory region allocated with their type. This means that at runtime we only need to test for NULL to ensure the program will not abort.

2.4 Specifications

There are four forms of specification in C0: pre- and postconditions, loop invariants, and assertions. All four forms consist of an expression of type bool that should evaluate to true at runtime; when this is the case we say that the specification *holds*. These expressions can be arbitrary C0 augmented with a few extra constructs, provided that the expression does not have visible side effects. Each type of specification is allowed only in certain contexts: pre- and postconditions on functions, loop invariants on while and for statements, and assertions effectively as program statements. All four start with //@, followed by an identifier for the type: requires, ensures, loop_invariant, and assert, respectively. Specifications followed by other code on a line must use the /*@...@*/ form, so that, syntactically, C0 specifications are ignored by a C compiler.

In addition to the full `C0` language, specifications have access to additional information not available to regular `C0` statements. In particular, the construct `\length` is available to determine the runtime length of an array. This allows the program to express an invariant that a loop counter stays within the bounds of an array, for example. In `//@ensures` clauses, the special variable `\result` is bound to the return value of the function. As an example, the absolute value function has a postcondition `//@ensures \result >= 0;`. There can be multiple specifications of a given type, in which case they mean their conjunction using `&&` in source order. Another way to write the postconditions for `func` in Figure 2.3 would be `//@ensures Q1 && Q2;`. For this reason, we can refer to "the" precondition or postcondition of a function or "the" invariant of a loop.

Pre- and postconditions allow students to express contracts on their code. For example, a function to search a linked list for a specified key would require that the linked list does not form a loop. It is a caller's responsibility to meet the requirements of the pre-conditions of a function just prior to the invocation of that function. Likewise, it is the called function's responsibility to ensure the postconditions hold immediately before the function returns to the caller. This pair of responsibilities form a contract between the caller and the callee: if the preconditions are met, then the function will execute successfully and the postconditions will be true.

This allows the program to be understood without looking at the body of the function; only the contract, its pre- and postconditions, need to be examined. This also means that we should be able to substitute any other function with the same contract in the place of a called function, such as a different implementation of a sort. Further, to understand a function's purpose, we do not need to know where it is called, because the required context is summarized in the precondition. In this sense, we should be able to substitute the surrounding program for another and our function should still execute correctly. This is a strong notion of encapsulation or modularity, and allows us to reason about the program one function at a time. This is good not only for understanding the program, but also for analyzing it.

While the contract specifications operate at the level of functions, both loop invariants and assertions apply to statements within a function. In a sense, they express a similar kind of contract, but at the intra-function level. A loop invariant gives the property that is preserved across any number of iterations of a loop. Combined with the exit condition for the loop, this allows the surrounding code to be understood without looking at the body of the loop.[6] Similarly, an assertion breaks the program in two: the part before the assertion, and the part after. The part before must establish, and the part after may assume it. The difference between both of these and contracts is that not all properties that the program relies on are written down in assertions or loop invariants, so there is no equivalent of the substitution property for functions.

[6]`C0` does not support variants, which are required to make this statement strongly. Most loops that students write have obvious variants. Missing invariants often allow the program to complete successfully with a subtly incorrect result, but failure to adhere to a variant is very observable: the program loops forever!

2.4.1 Data Structure Invariants

While assertions and loop invariants are important for understanding complicated algorithms, they are still local in that all of the relevant details are within one function. A data structure, by contrast, is intended to communicate information between different parts of the program. Because it could be modified in any number of places, it does not seem possible to reason about data structures locally like we can with functions. Many data structures have complex invariants which are easy to break through bugs in the modification code, such as a binary search tree remaining sorted, a red-black tree remaining balanced, or a doubly linked list being properly doubly linked. For this reason, many languages with contracts include special support for such invariants. `C0`, rather than providing another construct, relies on an idiomatic style of specification by writing *invariant functions*.

A data structure consists of the memory representation, such as a set of structs and arrays linked by pointers, an invariant function, (usually called `is_hashtable`, `is_list`, etc.), and some modification and query functions. The invariant function holds only when the internal invariants of the data structure are satisfied. For example, a linked list of positive integers has the invariants that it is a well-formed list, and also that each data element is greater than zero. Functions which modify the data structure would have this function as both a pre- and postcondition. For example, insert into a hash table would require the table to be inserted into `is_hashtable`, and it would ensure at the end that it `is_hashtable` again, and that the key actually was inserted.[7]

`C0`, like `C`, does not provide any encapsulation for structs in that there are no public or private fields. The notion of specification that is given by these data structure invariants, while not achieving information hiding *per se*, provide a better means of ensuring invariants are preserved than simply making the relevant fields or types private. The invariants must explicitly be written down rather than being informal, which assists with debugging. Sometimes it is necessary to partially or completely break the invariant for a data structure in the process of updating it. A common example is that while inserting into a self balancing tree, the tree is not balanced everywhere. This idiomatic way of specifying data structures causes the students to think explicitly about when in execution the invariant needs to hold, and when some parts of the invariant may be broken.

As an example, let us consider a binary tree in `C0`. We might use this data structure to implement a set of integers, in which case the tree should be a binary search tree, i.e. it should be sorted. The nodes will be structs, where `NULL` represents a leaf with no data:

[7] To complete the specification, we would have to say that nothing else was inserted and nothing has left the table, but that is difficult to write in the relatively impoverished `C0` language.

```
struct node {
   int data;
   struct node* left;
   struct node* right;
}
```

Then we might write `is_tree` as:

```
int max(tree t)
  //@requires t != NULL;
{
  int m = t->data;
  if (t->left != NULL) m = int_max(m, max(t->left));
  if (t->right != NULL) m = int_max(m, max(t->right));
  return m;
}
int min(tree t)
  //@requires t != NULL;
{
  int m = t->data;
  if (t->left != NULL) m = int_min(m, min(t->left));
  if (t->right != NULL) m = int_min(m, min(t->right));
  return m;
}
bool is_tree(tree root) {
  if (root == NULL) return true;
  return(root->left == NULL || max(root->left) < root->data)
     && (root->right == NULL || min(root->right) > root->data);
}
```

The `is_tree` acts as an invariant checker, in that we can execute it at runtime to verify that the tree is actually ordered. The insert function would have `is_tree` as both a precondition and a postcondition, as in these two places the tree should be sorted. This implementation of checking is not particularly efficient, but when writing specifications we are concerned with ease of understanding, and not runtime efficiency. This is because we only dynamically check contracts during development and testing of the program. Once we have confidence that the program is correct, the specification checking can be disabled to allow the program to run significantly faster. An advantage of adding specifications is that errors are detected as closely as possible to the place in the code where the error is introduced. If specifications are missing, bugs tend to manifest later in the execution than, and far away in the code from, the line which contains the error.

Specification functions are not supposed to have visible side effects, so writes to externally visible memory are not allowed. Assuming that no specification fails to hold in the program, it is thus safe to remove the specification checks once the program has been debugged. We are particularly interested in the case where all operations are safe, assuming the specifications all hold. In this case, we could remove even the checks mandated by `C0` and not change the meaning of a program. If this property were to hold for a `C` program, then it would be guaranteed not to evaluate undefined behavior. Because undefined behavior in `C` is so dangerous, we want to teach the students how to avoid it completely. It is with this goal in mind that we describe the main contribution of our work, the analysis itself.

Chapter 3

Analysis Framework

Our contribution is a tool which is integrated into the `C0` compiler as a separate compilation mode. Rather than producing an executable, when the `C0` compiler is invoked with the `-S` flag, the tool analyzes the input program to determine if there are safety violations, reporting errors for any violations found. Our tool is necessarily conservative, in that it may give an error for programs which will not go wrong at runtime. In other words, it is not complete. It is, however sound, in that if a supported construct is not flagged, then it will not cause a crash at runtime. This partial safety guarantee means that if there are no errors, then the program will not crash due to null pointer dereferences.

Internally, we split the tool into two phases: the first translates from the compiler's `C0` AST into a program in a simple intermediate language known as GCL (inspired by Dijkstra's Guarded Command Language [3]), and the second analyzes this program to determine which program points are unsafe. This approach factors the analysis problem into understanding the semantics of `C0` in the translation, and understanding the program logic in the analysis of GCL. This approach follows that of Boogie and other intermmediate verification languages [5]. In this chapter, we will present GCL, the translation from `C0`, and the generic analysis of GCL applicable to multiple kinds of analysis (such as for other safety properties such as array bounds checking, division by zero, etc).

3.1 Goals of our analysis

The goal of our analysis is to give immediate, useful feedback to students about their code. Because we want to encourage students to write more specifications, a program which operates correctly, but for which there are not sufficient specifications to reason locally about that correctness is not considered a correct program. We specifically target null pointer dereferences, so we do not handle integers, arrays, etc. Because we are not tracking this other information, it is impossible for our tool to be able to prove that all the specifications in the program hold. This means our objective is a bit different from most software verification tools: although we employ similar techniques to perform the analysis, we do not attempt to check whether the specifications themselves will always hold. This is similar to the approach of Hovemeyer et. al. [4], except in that work branch conditions were used in place of specifications, and the tool was not intended to be sound (i.e. catch all potential errors).

Our primary goal is to provide feedback to the students when they appear to have made a mistake, so that they can fix that mistake. Rather than run a test suite, which requires the student to make a context switch from reasoning abstractly about the program to tracing a concrete execution, the feedback from our tool remains at the level of the code on the screen. This entails a secondary goal, which is to make the analysis fast. We want to enable a tight interaction loop, where the student is able to make changes to their program (including adding new specifications) and immediately see the effects of those changes.

To meet our performance goal, we chose to make our analysis *local*, which also has the benefit of encouraging specifications. A typical static analysis tool looks across function boundaries, because the information it needs about a function must be inferred from how the function is used and defined. For this reason, such analyzes are called *whole program*, because they look at the entire program at once. By contrast, a local analysis considers only part of the program at a time; the rest of the program is considered unknown. Local analysis essentially breaks the program into pieces which are analyzed independently, which makes it significantly faster than an analysis which must track information across function boundaries.

For example, consider a function `pop` which crashes if its argument is a null pointer. If the program is correct, then every call to `pop` would be on a non-null argument. A whole program analysis would be able to inspect every call site, and see that it is always called on non-null arguments. A local analysis, by contrast, does not consider the call sites of `pop`, and so it is not able to determine this property. Thus in order for the local analysis to reason that a dereference in the body of `pop` is safe, this property must be stated in the contract of `pop`, i.e. a precondition must be present. We consider this an advantage of local analysis, as it essentially requires specifications to explicitly state important program properties.

One of the challenges is the fact that in `C0`, specification functions are written in the full `C0` language, with all the complexities of having full control flow available. We would like our analysis to be powerful enough to analyze specifications which encode complex logical properties, such as disjunctions and quantifiers. This would require in general inferring things such as loop invariants or recursive postconditions. If necessary, because the class is teaching the students how to write `C0` code, it is possible to have students write specification functions in a style amenable to analysis.

Finally, we would like the analysis to be simple, so that it is easy to explain to students why their program is or is not passing the tool. If students are confused by why the tool is passing or not passing their program, then the overhead of using the tool may outweigh the benefits.

3.2 GCL, a simplified `C0`

GCL is essentially a much stricter and more verbose `C0`. GCL is designed to make analysis easy by moving evaluation order from the level of expressions in `C0` to the level of statements in GCL. For example, in `C0`, we might write `if (a() && b()) {...}`, but in GCL this would be written:

```
b1 := a()
if(b1) {
   b2 := b()
   if(b2) { ... }
}
```

because in `C0`, `b` is only evaluated if `a` is true. There are additional complications with expressions such as `A[A[3+f()]] = g(A, 4)` in `C0`. The control flow of this expression is fairly complicated, and so in GCL the forms allowed on a left hand side of an assignment are very simple.

Further, expressions never cause the program to abort in GCL. For example, dereferencing a null pointer in GCL does not cause the program to go wrong, but instead returns an unspecified value. The translation has the property that all dereferences are immediately preceded by an assertion that the dereferenced pointer is not `NULL`. Expressions in GCL may depend on the state of the heap, but they may not modify any locations. Only assignments, either through function calls or the left hand side denoting a heap location, may modify the heap. These properties taken together mean that in GCL there is no need for a definition of evaluation order for expressions.

3.2.1 GCL Basics

GCL is oriented primarily around statements, summarized in Figure 3.1.

A program in GCL consists of a set of functions and struct definitions. The typedefs and `#use` directives in the `C0` program are expanded, so that the only top level definitions are structs and functions. Because GCL does not have a concrete syntax, forward declarations are not necessary. In GCL, nested structs are encoded by allowing GCL fields to be a sequence of `C0` fields. During the translation from `C0`, nested structs are expanded into such extended fields.[1] For example, `struct s` in GCL would have three fields: `i`, `f.g`, and `f.c` if the following definitions are from `C0`:

```
struct s {
   int i;
   struct t f;
}
struct t {
   int* g;
```

[1]In practice, this unwrapping code was only exercised by the compiler's regression suite.

```
stmt   ::=  if (e) {stmt} else {stmt}            (Conditional)
         |  while inv spec {stmt}                (Loop with invariant)
         |  assume e                             (Assumption)
         |  assert e                             (Assertion)
         |  lhs := rhs                           (Assignment)
         |  block int-literal {stmt}             (Define block)
         |  break int-literal                    (Break from block)
         |  stmt; stmt                           (Sequence)
         |  nop                                  (Nop i.e. skip)

rhs    ::=  e                                    (Expression)
         |  alloc(type) | alloc_array(type, e)   (Allocations)
         |  func(e, e, ...)                      (Mutating call)

lhs    ::=  ident                                (Variable)
         |  ident->field                         (Field dereference)
         |  *ident                               (Cell dereference)
         |  ident[ident]                         (Array assignment)

e      ::=  ident                                (Variable)
         |  op(oper, e, e, ...)                  (Builtin operation)
         |  e == e | e != e                      (Comparisons)
         |  e[e]                                 (Array access)
         |  e.f                                  (Field access)
         |  *e                                   (Pointer dereference)

field  ::=  ident | field.ident                  (Fields)
```

Figure 3.1: Syntax of GCL.

```
    char c;
}
```

A function in GCL consists of the signature, giving the number and types of its arguments and return type, its pre- and postconditions, and the function body. The pre- and postcondition are represented as triples of statements, one each for checking well-formedness, checking the specification holds, and assuming the specification holds. The reason we need all three of these is that when we check a function's body, we need to check that the precondition is well-formed in all possible contexts, without asserting it as true (which would nearly always spuriously fail). Then we need to assume that it has been established before we begin checking the function's body. A similar situation arises when we check a loop invariant: before the loop and after the body executes the analysis needs to check the specification, and after a generic loop iteration it needs to assume that the invariant holds. These specifications are represented as GCL statements, that when executed have the appropriate effect on the state of the program. For example, the precondition `//@requires p != null && p-> f != null;` would translate to

```
assert(p != null)
assert((*p).f != null)
```

to check the precondition at a call site, and

```
assume(p != null)
assume((*p).f != null)
```

for the assumption at the beginning of a function. When checking the body of a function, the well-formedness and assumptions from the precondition are needed and the well-formedness and assertion of the postcondition are needed. When analyzing a call to that function, the assertion from the precondition and the assumption of the postcondition are required. Thus a function in GCL contains all three elements from both the pre- and postconditions.

Statements in GCL are similar to `C0` or any simple imperative language, with the addition of the labeled break and block statements. In addition to the forms listed in Figure 3.1, each statement in GCL may have a label which allows additional information to be stored outside the AST. This is used, for example, to provide source region information, details about the origin and purpose of assertions, and the sets of variables and fields that a while loop writes to. Additionally, in the implementation, type information is embedded in the AST, which includes both a mapping of the local variables of a function to their types as well as embedding of the types of expressions in the AST nodes. Because the `C0` program itself has been type checked, this type information is readily available to be added to the GCL AST.

The first statement is the ever popular if, which has the same meaning as always: one of the two statements is executed depending on the value of the condition. An addition is that the condition can be the nondeterministic operator $\star$, which is only allowed as an if condition, as in `if(`$\star$`)`.... An if with nondeterminism can be combined with assume statements to simulate the effect of a conditional branch without requiring the condition to be an expression. The section on translation from `C0` explains why this is helpful.

Assignment allows updating of locals and heap locations, as well as function calls and allocating memory. The left hand side of an assignment is syntactically constrained so that heap locations are specified in a heap independent way. This is because in `C0` the right hand side may modify the heap (through an allocation or a call to a function) and should not change the location referred to on the left. This means there does not need to be a specified ordering on the evaluation, other than the write to the heap occurring after the right hand side. In full `C0`, some parts of evaluating the left occur before the right hand side, and some occur after, which complicates the semantics.

Function calls and allocations both use the assignment statement. The GCL type system is like that of `C0` without typedefs, but including void as a unit type (i.e. a type with only one value). A call to a void function in `C0` is encoded in GCL by assigning the result to a dummy variable of type void. Allocations are similar to function calls, but they must be different syntactically because they take a type argument. Like `C0`, the array allocation takes a length parameter.

The labeled block and break statements form pairs used to implement a form of structured control flow necessary for translating `C0`. A block statement merely executes its child statement. When a `break` i statement is executed, control flow jumps to the nearest enclosing `block` with the same label (there must always be such a block in a well-formed GCL statement). This functionality, along with the repetition provided by while loops and the conditional execution provided by if, is able to encode all the control flow of `C0`, including breaks and continue in loops, return, and the short circuiting evaluation of `&&` and `||`.

Assert and assume form the basis of checking specifications in GCL. Assert has the same meaning as it does in `C0`: the condition must be true when the statement is executed, otherwise the program aborts unsuccessfully. Regardless, after the execution of the assert, the condition can be assumed to hold. Assume is similar, except that the program does not abort abnormally if the condition does not hold. Assume essentially allows us to know that its condition holds "for free". One interpretation is that an assume terminates the program without error if the. In GCL, we do not inspect the body of the callee when analyzing a call site. We effectively treat function calls as black boxes, where all we know is that the postcondition of the function holds.

We have described GCL using terms like "execute" and "evaluate", as if GCL has an interpretation in terms of concrete execution traces. This is perhaps good for intuition, but it is not the interpretation we will give to GCL. Instead, we will describe its semantics formally in terms of the effect of symbolically executing a GCL program and tracking an approximation or abstraction of the set of possible states the program could be in. Before we consider this, which forms the basis of our generic analysis of GCL programs, we will present the translation from `C0` into GCL.

3.3 Translation from C0 to GCL

Most of the work of translating C0 to GCL is straightforward, because the languages correspond closely. The interesting parts are translating specifications into the three statements, translating expressions so that their side effects occur in the proper order, and translation of assignments.

3.3.1 Expression Translation

To translate expressions, we need to lift all side effecting operations to the statement level in GCL. We do this by defining a translation from expressions e in C0 to a pair $(\check{e}, \hat{e})$ consisting of a statement $\check{e}$ containing the effects of the expression and any definitions of temporary variables it needs, and an expression $\hat{e}$ which gives the value of the expression. The rules for the translation ensure that the statements from subexpressions in C0 are composed in the proper order. The rules are given in Figure 3.2.

In some of these rules, we capture the value to a local variable t (which is generated fresh for each expression), and use that variable for $\hat{e}$. We do this because use of $\hat{e}$ may separated from the execution of $\check{e}$, and thus $\hat{e}$ may be evaluated in a different heap from $\check{e}$. If we included heap dependent constructs like pointer or array dereference, then the value could incorrectly change as side effects of other expressions are executed. We translate allocations and calls as statements because GCL requires these to be *rhs*'s syntactically.

3.3.2 Condition Translation

Conditions for ifs are translated somewhat differently. The issue comes from the way the expression translation would translate the following program:

```
if(a && b) { T } else { F }
```

as:

$\check{a}$
if($\hat{a}$) {$\check{b}$}
if($\hat{a} \wedge \hat{b}$) { T } else { F }

This introduces infeasible paths into the program that would require tracking the value of $\hat{a}$ in order to eliminate. Effectively, without looking at the conditions, there are four possible ways the control flow could pass through the previous program: both conditions are true, neither is, only the first is, and only the second is true. However, considering the conditions reveals that if T is executed then $\check{b}$ must have been as well, The "only second is true" case is thus an infeasible path. This is problematic if the statement $\check{b}$ contains information that is needed by the analysis in evaluating T, such as a variable definition. The analysis would have to represent the fact that at the program point between the two if statements in the translation, $\hat{a}$ implies $\check{b}$ has executed. If $\hat{a}$ does not fall into the domain of our analysis, then out analysis would lose this conditional

e		$(\breve{e}, \hat{e})$
v	$\rightarrow$	$(\mathtt{nop}, v)$
n (1,2,3, ...)	$\rightarrow$	$(\mathtt{nop}, n)$
`true`	$\rightarrow$	$(\mathtt{nop}, \mathtt{true})$
`false`	$\rightarrow$	$(\mathtt{nop}, \mathtt{false})$
`'c'`	$\rightarrow$	$(\mathtt{nop}, \mathtt{'c'})$
`"str"`	$\rightarrow$	$(\mathtt{nop}, \mathtt{"str"})$
`NULL`	$\rightarrow$	$(\mathtt{nop}, \mathtt{NULL}_\tau)$
$a\mathtt{[}i\mathtt{]}$	$\rightarrow$	$(\breve{a}; \breve{i}; \mathtt{assert}\ 0 \leq \hat{i} < \mathtt{\backslash length}(\hat{a}); t\mathtt{:=}\hat{a}\mathtt{[}\hat{i}\mathtt{]}, t)$
$\mathtt{*}p$	$\rightarrow$	$(\breve{p}; \mathtt{assert}\ \hat{p} \neq \mathtt{NULL}_\tau; t\mathtt{:=*}\hat{p}, t)$
a `&&` b	$\rightarrow$	$(\breve{a}; \mathtt{if}(\hat{a})\{\breve{b}\}, \hat{a} \wedge \hat{b})$
a `||` b	$\rightarrow$	$(\breve{a}; \mathtt{if}(\neg\hat{a})\{\breve{b}\}, \hat{a} \vee \hat{b})$
c `?` a `:` c	$\rightarrow$	$(\breve{c}; \mathtt{if}(\hat{c})\{\breve{a}\}\mathtt{else}\{\breve{b}\}, \hat{c}\mathtt{?}\hat{a}\mathtt{:}\hat{b})$
$s\mathtt{.}f$	$\rightarrow$	$(\breve{s}, \mathrm{extend}(\hat{s}, f))$
$\mathtt{alloc}(\tau)$	$\rightarrow$	$(t\mathtt{:=alloc}(\tau), t)$
$\mathtt{alloc\_array}(\tau, e)$	$\rightarrow$	$(\breve{e}; t\mathtt{:=alloc\_array}(\tau, \hat{e}), t)$
`\result`	$\rightarrow$	$(\mathtt{nop}, \mathtt{\backslash result})$
$\mathtt{\backslash length}(e)$	$\rightarrow$	$(\breve{e}, \mathtt{\backslash length}(\hat{e}))$
$\mathtt{f}(e_1, e_2, ...)$	$\rightarrow$	$(\breve{e_1}; \breve{e_2}; ...; t\mathtt{:=f}(\hat{e_1}, \hat{e_2}, ...), t)$
a int-op b	$\rightarrow$	$(\breve{a}; \breve{b}; \mathrm{check}(\text{int-op}, \hat{a}, \hat{b}), \hat{a}\ \text{int-op}\ \hat{b})$
a cmp b	$\rightarrow$	$(\breve{a}; \breve{b}, \hat{a}\ \mathrm{cmp}\ \hat{b})$

Figure 3.2: The translation rules for expressions. int-op stands for any of the integer operations like addition and shift. cmp stands for any comparison. check gives the statements which should check that int-op is safe with arguments $\hat{a}$ and $\hat{b}$. For example, for division and modulus this is $\mathtt{assert}\ \hat{b} \neq 0 \wedge (\hat{a} \neq \mathtt{MIN\_INT} \vee \hat{b} \neq -1)$.

C(`true`, T, F)	=	T
C(`false`, T, F)	=	F
C(a `&&` b, T, F)	=	C(a, C(b, T, F), F)
C(a `||` b, T, F)	=	C(a, T, C(b, T, F))
C(`!`a, T, F)	=	C(a, F, T)
C(c `?` a `:` b, T, F)	=	C(c, C(a, T, F), C(b, T, F))
C(e, T, F)	=	$\check{e}$`;` `if`($\hat{e}$, T, F)

Figure 3.3: Translation rules for boolean expressions as conditions.

information and spurious errors would be generated. To eliminate this problem, we could translate the original if instead as:

```
ǎ
if (â) {
  b̌
  if(b̂) { T }
  else { F }
} else { F }
```

but this duplicates the statements `T` and `F`, which can lead to an exponential blowup in the size of the program. If we had `goto`, then we could just define two labels and jump to the appropriate one in the leaf branches of the decision tree. GCL does not include `goto`, but it turns out that the `block`/`break` construct is sufficient to handle this. To translate a condition, we need two GCL statements: one to execute when the condition is true, and the other when it is false. We use the notation C(e, T, F) to mean the statement which executes T if e is true and F otherwise.

The rules for the various boolean cases are depicted in Figure 3.3. We make use of this translation when we translated if statements in the next section. By using block/break appropriately in that translation, T and F will always be just constant sized break statements, rather than the translation of the original true and false branches of the if.

3.3.3 Statement Translation

Some of the statements in `C0` have no equivalent in GCL, or their forms have been significantly changed. Here we will explain how each `C0` construct is encoded into GCL.

1. Assignment. Assignment is similar to `C0`, except that various expressions need to be pulled out of the assignment in order to guarantee the proper order of evaluation defined by `C0`. In GCL, all locals are effectively declared at the top of a function, so variable definitions in `C0` become simple assignments to their initializing expression in GCL.[2] In order to match the

[2]Variables in `C0` are required to be initialized before use, which is checked by the main compiler during type checking.

semantics of `C0` given in Section 2.3.3, the translation rules for l-values match the rules for the equivalent expressions, except the form `*lv` needs its nullity check moved after the evaluation of both the left and right side effects. Because of the syntactic restriction of l-values in GCL, the translation captures the l-value up to the final dereference or array access into a variable. This dereference or access then becomes the final assignment, after the right hand side effects are executed. As described previously, the check for nullity of the left hand side in the case of writing to a pointer is performed immediately before the assignment, after the right hand side execution.

2. If statements. The translation of if statements is complicated by a desire to not introduce spurious parallel ifs which would otherwise be generated by the condition translation. We have already seen how a boolean expression can execute arbitrary statements depending on the result. We would like to pass `goto`s to this translation to avoid duplicating the body of the if, but `goto` is not supported. Instead, for an if statement `if(`e`)` T `else` F, we generate new block labels E_L, T_L and F_L, let $\bar{e} = C(e, \texttt{break}\, T_L, \texttt{break}\, F_L)$, and generate:

```
block $E_L$ {
  block $F_L$ {
    block $T_L$ {
      $\bar{e}$
    } $T$; break $E_L$
  } $F$
}
```

   If the condition is true, then we break out of the innermost T_L block, and thus execute the T statement. Breaking out of E_L after T ensures we do not execute F. If the condition is false, then we break out of F_L, skipping over T, and jumping to F. Control then passes out of E_L and to the rest of the program. The translation C has the property that the last statement executed along any control flow path is one of the two breaks. We could therefore omit the T_L block and break pair, allowing control flow to continue directly to T after $\bar{e}$ on the true branch. We avoid relying on this property, however, and use this more verbose but also more symmetric translation.

3. Expression statements. An expression statement is a bare expression by itself in `C0`, which in reasonable code is almost always a function call. These are translated by taking $\check{e}$ and $\hat{e}$, and only keeping $\check{e}$. For functions, this works because $\check{e}$ always contains the function call itself, assigned to a variable (as is required by GCL syntax).

4. While. A while statement in `C0` consists of an invariant, a loop test, and a body. We translate a loop:

```
while(e) //@loop_invariant I;
{ B }
```

into:

```
block BREAK {
  while inv I {
    if (!e) break BREAK;
    B
  }
}
```

We reuse the conditional evaluation code, passing `nop` as the true action and `break BREAK` as the false action. `BREAK` stands for a integer constant which is used for all while loops. To support continue, we could wrap the body in a block CONTINUE and turn `continue` into `break CONTINUE`. However, the use of break and continue is restricted to the extension `C1`.

5. Return. There are two forms for return in `C0`, depending on whether the containing function returns void. If there is a value to return, then we translate return to `\result := e; break RETURN`. The body of every function in GCL is wrapped in a `block RETURN` by the translator, so that the return breaks cause the program to jump to the end of the function. Effectively, during checking the postconditions are checked by placing the assertion statements immediately after this function body block.

6. For, increment, decrement. These expression forms have already been elaborated into other forms by the main `C0` compiler, as described in Section 2.3.3.

7. Assertions. Both `//@assert` and `assert()` are turned into assert(s) in GCL. The former uses the specification translation, given below (the well-formedness check followed by the assert true), and the later uses the expression translation to get a translation for the condition, which is then asserted to be true.

3.3.4 Specification Translation

We define the translation of `C0` specifications using an auxiliary transform that produces 5-tuples of GCL statements:

$$\mathrm{S}(e) = (S_{\mathrm{wf}}(e), S_{\mathrm{asrt}}(e), S_{!\mathrm{asrt}}(e), S_{\mathrm{asm}}(e), S_{!\mathrm{asm}}(e))$$

We let S_{wf}, S_{asrt}, $S_{!\mathrm{asrt}}$, S_{asm}, $S_{!\mathrm{asm}}$, be the projections from this tuple. S_{wf} is the statement which checks well-formedness. S_{asrt} and $S_{!\mathrm{asrt}}$ check if the specification is true or false respectively. S_{asm} and $S_{!\mathrm{asm}}$ have the effect of assuming that the specification is true or false. By adding the false forms $S_{!\mathrm{asrt}}$ and $S_{!\mathrm{asm}}$ to the translation, we can easily handle logical negations (`!`). Due to the symmetry of boolean logic, the translation with these extra elements is not much more complicated than without them. The three components that are required elsewhere in the translation are S_{wf}, S_{asrt}, and S_{asm}; the false forms are only required for the recursive definition of S.

The reason we do not just reuse the conditional translation is that this would produce much less understandable GCL output. For example, the expected translation of `//@assert a && b;` is something akin to:

```
assert(a)
assert(b)
```

rather than:

```
if(a) {
   if(b) { assert true }
   else { assert false }
} else { assert false }
```

even through they have the same meaning. Further, this translation lets us give better error messages, because for example we can identify the failing conjunct when an assertion may not hold.

The rules for the translation are given in Figure 3.4. Most rules are fairly straightforward, but we will cover some in more detail. The rules for a `&&` b and c `?` a `:` b explain why we need $\star$ as a nondeterministic operator. Because the specification translation does not give us an expression, we need to use the two assumption statements computed from the subexpressions to get the same effect. We could use the `?:` to translate `&&` like we use it for `||`, but by special casing it in the translation we generate the much cleaner $S_{\text{asrt}}(a); S_{\text{asrt}}(b)$ for a `&&` b for this extremely common case.

Function calls inside specifications are handled a bit differently from function calls in the expression translation. Because these functions are pure and cannot affect the heap, we can use them as expressions in GCL. Rather than moving these to the top level, as in:

```
b := is_tree(t)
if (b)
   ...
```

which would require tracking the value of b and the consequences if it is true or false between the two statements, this may be translated as:

```
_ := is_tree(t)
if(is_tree(t))
   ...
```

Because the only side effects a call may have in a specification is aborting due to its own internal specification failure, all we need to do is check the well-formedness of the call by calling it normally.

In translating a general expression e, which includes function calls, we need to treat all assertions as assumptions for S_{asm} and $S_{!\text{asm}}$. A general expression e

$$
\begin{array}{lcl}
S(\texttt{true}) & = & (\texttt{nop}, \texttt{nop}, \texttt{assert false}, \texttt{nop}, \texttt{assume false}) \\
S(\texttt{false}) & = & (\texttt{nop}, \texttt{assert false}, \texttt{nop}, \texttt{assume false}, \texttt{nop}) \\
\\
S_{\text{wf}}(a \;\texttt{\&\&}\; b) & = & S_{\text{wf}}(a); \texttt{if}(\star)\{S_{\text{asm}}(a); S_{\text{wf}}(b)\}\texttt{else}\{\,\} \\
S_{\text{asrt}}(a \;\texttt{\&\&}\; b) & = & S_{\text{asrt}}(a); S_{\text{asrt}}(b) \\
S_{\text{!asrt}}(a \;\texttt{\&\&}\; b) & = & \texttt{if}(\star)\{S_{\text{asm}}(a); S_{\text{!asrt}}(b)\}\texttt{else}\{S_{\text{!asrt}}(a)\} \\
S_{\text{asm}}(a \;\texttt{\&\&}\; b) & = & S_{\text{asm}}(a); S_{\text{asm}}(b) \\
S_{\text{!asm}}(a \;\texttt{\&\&}\; b) & = & \texttt{if}(\star)\{S_{\text{asm}}(a); S_{\text{!asm}}(b)\}\texttt{else}\{S_{\text{!asm}}(a)\} \\
\\
S_{\text{wf}}(c \;\texttt{?}\; a \;\texttt{:}\; b) & = & S_{\text{wf}}(c); \texttt{if}(\star)\{S_{\text{asm}}(c); S_{\text{wf}}(a)\}\texttt{else}\{S_{\text{!asm}}(c); S_{\text{wf}}(b)\} \\
S_{\text{asrt}}(c \;\texttt{?}\; a \;\texttt{:}\; b) & = & \texttt{if}(\star)\{S_{\text{asm}}(c); S_{\text{asrt}}(a)\}\texttt{else}\{S_{\text{!asm}}(c); S_{\text{asrt}}(b)\} \\
S_{\text{!asrt}}(c \;\texttt{?}\; a \;\texttt{:}\; b) & = & \texttt{if}(\star)\{S_{\text{asm}}(c); S_{\text{!asrt}}(a)\}\texttt{else}\{S_{\text{!asm}}(c); S_{\text{!asrt}}(b)\} \\
S_{\text{asm}}(c \;\texttt{?}\; a \;\texttt{:}\; b) & = & \texttt{if}(\star)\{S_{\text{asm}}(c); S_{\text{asm}}(a)\}\texttt{else}\{S_{\text{!asm}}(c); S_{\text{asm}}(b)\} \\
S_{\text{!asm}}(c \;\texttt{?}\; a \;\texttt{:}\; b) & = & \texttt{if}(\star)\{S_{\text{asm}}(c); S_{\text{!asm}}(a)\}\texttt{else}\{S_{\text{!asm}}(c); S_{\text{!asm}}(b)\} \\
\\
S(a \;\texttt{||}\; b) & = & S(a \;\texttt{?}\; \texttt{true} \;\texttt{:}\; b) \\
S(\texttt{!}a) & = & (S_{\text{wf}}(a), S_{\text{!asrt}}(a), S_{\text{asrt}}(a), S_{\text{!asm}}(a), S_{\text{asm}}(a)) \\
S(e) & = & (\check{e}, \check{e}; \texttt{assert}\ \hat{e}, \check{e}; \texttt{assert}\ \neg\hat{e}, \\
 & & \quad e_{\text{asm}}; \texttt{assume}\ \hat{e}, e_{\text{asm}}; \texttt{assume}\ \neg\hat{e}) \\
 & & \text{where } e_{\text{asm}} = \check{e}[\texttt{assert} \to \texttt{assume}]
\end{array}
$$

Figure 3.4: Translation rules for specifications.

translates to an effect $\check{e}$ and a value $\hat{e}$. The meaning of $S_{\text{asm}}(e)$ is that it assumes that e evaluated successfully and it was true. For example, if e is `a[i]` (so `a` is an array of booleans), then if we assume e evaluated to true, we know that `a` contains `true` at `i`, but we also know that $0 \leq$ `i` $<$ `\length(a)`. Similarly, if e was `p->f`, then we know `p` $\neq$ `NULL` in addition to the fact that `p->f` is `true`. We let e_{asm} be the statement $\check{e}$, except all assertions are instead assumptions. This is simple for statements `assert` and `assume`. The only caveat is that function calls also need to have the preconditions turned into assumptions as well. If we execute `assume` $f(a_1, ...)$, then we assume the preconditions and postconditions of f held, and further that it returned true. Thus the well-formed check calls `_ :=`$f(a_1, ...)$ are entirely redundant, and so they are removed when we are generating e_{asm}. This also prevents the analysis from incorrectly generating errors about preconditions not holding.

3.4 Semantics of GCL

To analyze a GCL program, we need to first consider the semantics of GCL. We begin by observing that the state of a GCL program is a subset of the state of a `C0` program at a particular program point. GCL has, at the minimum, the same variables, and the heap at any point is the same shape.[3] The difference is that

[3]Ignoring the differences in how nested structs are handled. These structs are equivalent in `C0` and GCL because of the restrictions on large types, which means accesses to sub-structs

GCL program points may correspond to the middle of expressions or statements in `C0`. This state correspondence means that if we determine that a property is true of the GCL state at a particular program point, then that property will be true at the corresponding `C0` program point. In particular for our contribution, this means that a pointer demonstrated to be non-null in GCL will be so in `C0` as well.

Because we are interested in a sound analysis tool, we must consider every possible execution of the program. We do this by giving the semantics of a GCL statement as a pair of the effect it has on the state of the program and the possible errors it may run into during execution. In particular, we let a *concrete state* be the state of the local variables and the heap. A GCL program, like a `C0` program, can run on specific, concrete inputs. In GCL, computation is allowed to branch nondeterministically, so a single concrete state may become multiple possible concrete states. In addition, a GCL statement may either continue execution at the next statement or break out of an enclosing block. We call a set of states a *context*. In general then, a GCL statement takes a pre-context from before the statement to a post-context and a mapping from block indices to contexts, while errors are generated about the assertions that may not hold within the statement.

With this interpretation we can easily see how to perform an analysis. We define a context which upper bounds the possible concrete states, using some approximation. Then we run our analysis for each statement to obtain a context for each program point. An assertion is guaranteed to hold if it holds in every concrete state in the context. We can check if assertions are possibly violated by checking the condition of the assertion in every concrete state. Of course, the number of concrete states is unbounded in general, and reasoning about them exactly would be equivalent to the halting problem. Instead, we choose a particular representation which captures the properties that we are interested in analyzing. In Chapter 4, we present one such representation for pointer analysis.

Because a context in general is a bound on the possible states, we can treat them as elements of a bounded lattice, with the ordering operation being ordinary set inclusion, the empty set as the bottom element and the top element as all possible states. The least upper bound is set union and the greatest lower bound is intersection. While each concrete state is finite, the elements of the lattice typically contain infinitely many concrete states. Again, because representing the exact reachable states is impossible, we compute a bound on the "true" context.

Before we consider how each GCL statement changes a context, we call special attention to the two related statements `assert` and `assume`. The `assert` statement is the only way a GCL program can encounter an error. In `C0`, an error causes the program to abort. In GCL, execution is abstract, with a set of individual states, so the program does not terminate on encountering an error. If there are states in the pre-context in which the condition does not hold, then the program has an error, and these failing states are not in the post-context.

must occur as a syntactic series of field accesses.

In a sense, these individual states which did not meet the condition aborted abnormally, and the GCL program overall has an error if any individual state aborts. If the condition does not hold in any individual state then the post-context is the bottom context, i.e. the empty set. Because an assert does not modify any variables or the heap, no individual states are modified, i.e. the only effect on the context is removing states.

`assume` has a similar effect on the context by throwing out any states which do not satisfy the condition. For assumptions, the fact that the condition may not hold in some initial state(s) does *not* cause an error. Like `assert`, individual states which do not satisfy the condition terminate, but for `assume` they terminate *successfully*. The effect on the post-context is the same for both statements; the only difference is that `assert` may generate errors.

We can achieve this common effect of removing states which do not satisfy the condition using the greatest lower bound. We construct a context which is the set of states satisfying the condition. Then the result after the `assume`/`assert` is just the intersection of the initial context with this condition context. In lattice terms, we have taken the greatest lower bound of these two contexts. In logical terms, we have done something akin to "and": we have taken the states which are both in the pre-context *and* satisfy the condition.

3.5 Modified Fields Analysis

In order to analyze certain program statements, such as loops and function calls, we need to know an upper bound on the fields which these statements may modify. For example, if a callee modifies the `next` and `prev` fields on a doubly-linked list, then the caller must assume that information known before the call about those fields may no longer be true after the callee returns. Computing a tight set of modified fields would require running modification analysis concurrently with the central pointer analysis. Instead, we run an initial analysis pass which computes $\mathrm{mod}(f)$ for functions f and $\mathrm{mod}(s)$ for while statements s by observing syntactic field assignments and reachability of structs from the arguments to a function. Because the programs that students write rarely rely on specific preservation of this information across calls, this coarse analysis is sufficient in practice.

The analysis relies on the fact that a function can only modify fields which are reachable from its arguments, and can only modify fields which it syntactically writes to or which are modified by its callees. We define the notion of reachability $\mathrm{reach}(s,t)$ to mean given a access to a struct s, then we may obtain access to a struct t. (Access my come from arrays, pointers, or by embedding the struct in another.) The defining rules for $\mathrm{reach}(s,t)$ are:

$$\frac{\mathrm{reach}(x,y) \quad \exists \mathit{sf} \in \texttt{struct}\ y.\ \texttt{struct}z \in \mathrm{typeof}(\mathit{sf})}{\mathrm{reach}(x,z)} \qquad \frac{}{\mathrm{reach}(x,x)}$$

The existential in the first rule essentially means that `struct` y has a field *sf* which contains `struct` z somewhere in its type, like the types `struct z*`, `struct z**`, `struct z[]*`, and `struct z`.

Next we define the notion of internal modification, as in when a function modifies a field in its body:

$$\frac{\_\,\texttt{.}\,sf\,\texttt{=}\,\_\ \in f}{\text{mod-int}(sf, f)} \qquad \frac{g\texttt{(...)}\ \in f \quad \text{mod}(sf, g)}{\text{mod-int}(sf, f)}$$

The first rule says a syntactic modification of *sf* is internal modification, and the second says that if g modifies *sf*, so does f, as long as g is a callee of f.[4]

Before finalizing the analysis, we must define the notion of a function reaching a struct through its arguments:

$$\frac{\text{reach}(a, s) \quad \exists v \in \text{args}(f).\ \texttt{struct}\ a \in \text{typeof}(v)}{\text{arg-reach}(f, s)}$$

Finally, we can define the main mod() rule:

$$\frac{\text{mod-int}(sf, f) \quad sf \in \texttt{struct}\ s \quad \text{arg-reach}(f, s)}{\text{mod}(sf, f)}$$

We can execute these rules to saturation using bottom up logic programming, because there are only a finite number of fields and functions in the program. Effectively, we keep adding terms like mod, reach, etc. until no rule matches any existing terms in such a way as to produce a new fact. We use all the fields appearing in terms mod(sf, f) which hold after the rules have run to completion as the set of fields that function f may modify. We refer to this set as mod(f). In addition, we can determine which mod-int facts apply to each statement, as they are calculated by which syntactic terms appear in the code. Thus we can compute a set mod(w) for each while statement in the program. In addition to these computed fields, we add all local variables assigned to in the body to the modification set for while loops.

3.6 Analysis of GCL Statements

Once we have fixed a particular representation for the contexts, we can define the operation of the analysis. The analysis needs a few operations from the context:

1. Test for bottom, $c = \bot$. This operation decides whether a context represents bottom, i.e. that the context is the empty set, implying the corresponding program point is unreachable.

2. Least upper bound, $c_1 \vee c_1$. This operation joins two contexts, giving a context which represents the union of the states in either. This is used at join points in the control flow graph. We also do not require the true least upper bound, but the closer we are to the true least upper bound, the less imprecision we will introduce.

[4]As specifications may not modify memory, calls in specifications are ignored.

3. Analyze assignment, $c[lhs := rhs]$. This gives the new context after an assignment is executed. This could involve a heap assignment, or an update to a variable.

4. Analyze conjunction, $c \wedge e$. Given a context c, this operation gives a context after the statement `assume` e is executed.[5]

5. Havoc part of the context, `havoc` $v, f, \ldots$ `in` c. This operation nondeterministically assigns to part of the state, including variables and fields. Effectively, this causes the analysis to "forget" any information it had about those locations, because after havocking, those locations could have any value. This is used to model the effects of loops and function calls. Because we treat a function call or a loop as a black box except for its contract, we remove any information we had about parts of the state that these constructs could modify. Then we can assume that the postcondition for functions or the invariant for loops holds.

With these operations defined for a context, we can analyze a GCL statement. Our analysis takes a initial context, and produces a context bounding the post states, a mapping from indices to contexts, and a set of errors produced by failed assertions within the statement. We write $A(c, s)$ for the post-context of s starting from context c. We write $B(c, s)$ to denote the set of break indices and contexts, and $E(c, s)$ to denote the errors generated by s starting in c. The rules for the various GCL statements are given in Figure 3.5.

Several of these rules need some explanation. First we see that `assume` and `assert` differ only in that assertion produces an error if assuming the opposite does not generate the bottom context. The ErrorOf function gives us the error message associated with the assertion via its label. This message is set by the translation, and varies based on why that condition was generated. This is the only way errors are introduced; in all other cases they are just collected from the rest of the analysis.

The rules for `break` and `block` interact to ensure that the contexts resulting from a `break` are collected by the nearest enclosing `block`. The context after a block is the least upper bound of the normal end state of the enclosed statement and all the breaks of the same index. The rule for B of a block ensures that these contexts are used in only one block, and the translation ensures all breaks match some enclosing block.

The rules for `while` are at first glance a bit strange. First, we observe the A-rule for `while` implies that a while loop never exits normally, because while loops are effectively `while(true)` loops in GCL. The translation inserts a block/break pair to handle the loop test condition from `C0`. The interpretation of a while loop in GCL is that first we need to establish the loop invariant on entry ($E(c,$ `assert` I$)$). Then we havoc the modified part of the state to simulate being in an arbitrary iteration of the loop (`havoc` $\mathrm{mod}(w)$ `in` c). Then we assume the

[5]In the implementation, contexts for both the assumption being true and false are calculated simultaneously.

$$
\begin{array}{lcl}
A(c, \texttt{if}(e)\ s_1\ \texttt{else}\ s_2) & = & A(c \wedge e, s_1) \vee A(c \wedge \texttt{!}c, s_2) \\
A(c, \texttt{while inv}\ \mathrm{I}\ s) & = & \bot \\
A(c, \texttt{assume}\ e) & = & c \wedge e \\
A(c, \texttt{assert}\ e) & = & c \wedge e \\
A(c, lhs \texttt{ := } rhs) & = & c[lhs \texttt{ := } rhs] \\
A(c, \texttt{block}\ i\ s) & = & \vee(\{A(c, s)\} \cup \{c \,|\, (j, c) \text{ in } B(c, s) \wedge i = j\} \\
A(c, \texttt{break}\ i) & = & \bot \\
A(c, s_1\texttt{;}\ s_2) & = & A(A(c, s_1), s_2) \\
A(c, \texttt{nop}) & = & c
\end{array}
$$

$$
\begin{array}{lcl}
B(c, \texttt{if}(e)\ s_1\ \texttt{else}\ s_2) & = & B(c \wedge e, s_1) \cup B(c \wedge \texttt{!}c, s_2) \\
B(c, \texttt{while inv}\ \mathrm{I}\ s) & = & B(\texttt{havoc}\ \mathrm{mod}(w)\ \texttt{in}\ c, \texttt{assume}\ \mathrm{I}\texttt{;}\ s\texttt{;}\ \texttt{assert}\ \mathrm{I}) \\
 & & \quad \text{where } w = \texttt{while inv}\ \mathrm{I}\ s \\
B(c, \texttt{assume}\ e) & = & \{\} \\
B(c, \texttt{assert}\ e) & = & \{\} \\
B(c, lhs \texttt{ := } rhs) & = & \{\} \\
B(c, \texttt{block}\ i\ s) & = & \{(j, c) | \, (j, c) \text{ in } B(c, s) \wedge i \neq j\} \\
B(c, \texttt{break}\ i) & = & \{(i, c)\} \\
B(c, s_1\texttt{;}\ s_2) & = & B(c, s_1) \cup B(A(c, s_1), s_2) \\
B(c, \texttt{nop}) & = & \{\}
\end{array}
$$

$$
\begin{array}{lcl}
E(c, \texttt{if}(e)\ s_1\ \texttt{else}\ s_2) & = & E(c \wedge e, s_1) \cup E(c \wedge \texttt{!}c, s_2) \\
E(c, \texttt{while inv}\ \mathrm{I}\ s) & = & E(c, \texttt{assert}\ \mathrm{I}) \\
 & & \cup E(\texttt{havoc}\ \mathrm{mod}(w)\ \texttt{in}\ c, \texttt{assume}\ \mathrm{I}\texttt{;}\ s\texttt{;}\ \texttt{assert}\ \mathrm{I}) \\
 & & \quad \text{where } w = \texttt{while inv}\ \mathrm{I}\ s \\
E(c, \texttt{assume}\ e) & = & \{\} \\
E(c, \texttt{assert}\ e) & = & \text{if}(c \wedge \texttt{!}e = \bot) \text{ then } \{\} \text{ else } \mathrm{ErrorOf}(\texttt{assert}\ e) \\
E(c, lhs \texttt{ := } rhs) & = & \{\} \\
E(c, \texttt{block}\ i\ s) & = & E(c, s) \\
E(c, \texttt{break}\ i) & = & \{\} \\
E(c, s_1\texttt{;}\ s_2) & = & E(c, s_1) \cup E(A(c, s_1), s_2) \\
E(c, \texttt{nop}) & = & \{\}
\end{array}
$$

Figure 3.5: Analysis rules of statements in GCL. In the rules for `while`, w stands for the whole while statement, i.e. mod is defined on the label for the overall while statement.

loop invariant, use this context to analyze the body, and finally check the loop invariant has been re-established by the loop body (`assume` I; s; `assert` I). When we write `assert` I, we mean the well-formed check and the assertion check from the specification translation, and when we write `assume` I, we mean the assumption translation. Thus I is technically a specification triple rather than a GCL expression.

3.7 Analysis of GCL Programs

A program has more than just statements in it; functions may call one another. Because our analysis is local, we treat a function call as a black box, keeping only the information we learn from the pre- and postconditions. As far as our analysis is concerned, only one function exists at a time. Thus in GCL we do not define a notion of call stack or call graph.

Analysis of a function itself begins with the context which represents all possible states, i.e. the top element of the lattice. Then we check the well-formedness of the preconditions, and then assume they are true. Then we analyze the body of the function. Because the translation wrapped the body in a block broken out of by `return`, the context after the body is the context representing all the states just after returns throughout the function body. Then we check that the postconditions are satisfied.

If a function is used in a specification, then we can use this final context as a *function summary.* It does not make sense for a specification function to repeat the conditions that it checked in the postcondition of the function. The context contains all the information that was known at each return site in the function. As a special case, when analyzing a specification function, we keep the contexts that hold when the return value is true separate from those in which the return value is false. This lets the summary be used at a call site using only the contexts which match the actual return value.

If specification functions are recursive or mutually recursive, then the analysis might encounter a call to a function which has not yet been summarized. In this case we cannot get any information from the call. To handle this case, we run the analysis a fixed number of times ($k = 2$). When we rerun the analysis, the callee will have been analyzed, and thus will have some summary. We cannot run to convergence because our analysis does not produce bounded answers. For the programs we investigated, this limitation did not affect the analysis because the functions were written without recursion. Certain tree like data structures have natural invariants written as recursive functions, but these were not present in the programs we considered.

To handle a function call, we first generate a new local variable $t_1, ..., t_n$ for each argument, and t_{call} for the result of the call. Then, we add the fact that each new local is equal to the corresponding argument supplied by the call site. We then rewrite the preconditions and postconditions with these new variables substituted for the original formal parameters of the callee. Then we assert that the precondition is satisfied, havoc the modification clause of the callee, and finally assume the postcondition. We generate new local variables so that we do not incorrectly change information about the formal parameters of the caller function which happen to have the same name as those of the callee. In effect, we transform a call v `:=` $f(a_1, ..., a_n)$ to a function with definition:

```
$\tau$ $f$($\tau_1$ $v_1$, ..., $\tau_n$ $v_n$) { ... }
```

into the following:

```
$t_1$ := $a_1$
⋮
$t_n$ := $a_n$
assert $\text{pre}_f[v_1, ..., v_n \leftarrow t_1, ..., t_n]$
havoc $\text{mod}(f)$
assume $\text{post}_f[v_1, ..., v_n, \texttt{\textbackslash result} \leftarrow t_1, ..., t_n, t_{\text{call}}]$
$v$ := $t_{\text{call}}$
```

Chapter 4

Pointer Analysis

In order to use the analysis of GCL programs to generate errors, we need to determine how we will bound the possible states in a context. Because we are focusing on null pointer dereferences, ultimately we are interested in showing that assertions of the form:

```
assert p != NULL
```

are satisfied. This means we want a variant of *alias analysis*. While there are many choices of potential analyses, we have a particular constraint in that we must be able to take advantage of information in the program of the form `p != q`. This might come through the condition on an `if`, or through a precondition. Many alias analyzes based on computing which structs *may-alias* are not capable of taking advantage of this kind of information.

This disequality is particularly necessary when we are trying to perform updates to the heap: if we do not track this information, then every heap update would throw out everything known about that field for other structs. In addition, we need to analyze code such as `node n = head->next`, so we have to be able to represent `p == q` to know that `n` and `head->next` point to the same location.

4.1 Context Representation

Based on these observations, we chose to represent a bound on states as a set of equalities and disequalities between *chains*. A chain is either `NULL`, a variable, or a chain dereferenced at a particular field (c`->`f). Because we only track chains with pointer type, we need only `->` and not the `*p` or `e.f` forms. A chain effectively refers to or labels a *location*, a place which stores a value. Each field in a struct and local variable is a location.[1] An advantage of chains is that they have an immediate meaning in the program, as they are expressions in `C0`. A

[1]Array elements and cells are locations too, but we do not track information about these locations.

$$
\begin{array}{rcl}
e & ::= & \texttt{NULL} \\
 & | & v \\
 & | & e\texttt{->}f \\
 \\
c & ::= & \bot \\
 & | & \vee(c_1, c_2, ...) \\
 & | & \text{Shape}(e_1 \mapsto e_2, ..., [e_i \neq e_j, ...])
\end{array}
$$

Figure 4.1: Representation of the pointer analysis context c and chains e.

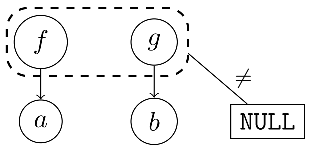


Figure 4.2: A simple shape description depicted graphically, representing the facts a`->`$f = b$`->`g and a`->`$f \neq$ `NULL`. The chains are a, a`->`f, b, b`->`g, and `NULL`. a, b, and `NULL` are in equivalence classes by themselves, while a`->`f and b`->`g are in the same class.

state will be in our set if all of the individual facts (equalities and disequalities) in our context are satisfied. By considering conjunctions of equalities and disequalities, we can represent the shape of the heap and the local variables.

It is not always the case that the expressions are syntactically chains. For example, we could have in the `C0` source program a comparison `a[3] != NULL`, which is not in chain form. In this case, we are not able to add any information to the context. In our corpus, there were no instances of a pointer being stored in an array, so it is not important for us to handle this case. The code that students write only uses structs which are pointed to by either variables or some other struct, and does not use arrays of structs or arrays of pointers to structs. Thus we can ignore expressions that are not syntactically chains and not lose precision over our corpus. To increase the flexibility of this representation, we extend this representation in two ways. The first is that we allow a separate representation for bottom and disjunctions of these shapes. This makes querying for bottom trivial, and allows us to handle divergent control flow.

The second extension is that when in specification functions, we keep separate contexts in which `\result` is true from those in which `\result` is false. Essentially we track the value of this one special boolean variable in the context, along with pointer information. The summary then becomes a pair of contexts, one for the summary which holds when the function returns true and one which holds when it returns false. This separation allows the analysis to easily handle statements like `assume` $f(...)$, which are critical to analyzing the idiomatic data structure invariant specifications.

At a concrete level, the context is representation is given in Figure 4.1. We represent the equalities using a map from chains to a representative for their equivalence class, as in the union-find data structure. Disequalities are a list of pairs of representatives.

These shape descriptions can be represented graphically, as in Figure 4.2. The chains are a, a`->`f, b, b`->`g, and `NULL`. We say that a is a subchain of a`->`f, and likewise for b and b`->`g. In the graphical representation, the node f represents the chain a`->`f. The arrows in this diagram are not the same as the mappings $e_1 \mapsto e_2$ given in . Those mappings represent the equivalence classes of the context, which the diagram represents as dashed region(s). In this example, a`->`f and b`->`g are in the same equivalence class. Finally, the line between the class and `NULL` represents the disequality a`->`$f \neq$ `NULL`. Such disequalities hold between *equivalence classes*, not individual chains. However, because we picked a representative from the class, we write these disequalities concretely as holding between these chains.

The invariants of the concrete representation are:

1. The least upper bound ($\vee$) is always the outermost form if present, and must have at least two subcontexts. The context can be rewritten:

$$\vee(\vee(a, b), ...) \rightarrow \vee(a, b, ...)$$

   because $\vee$ is associative. We can further rewrite $\vee(a) \rightarrow a$ and $\vee() \rightarrow \bot$.

2. No subcontext of $\vee$ may be bottom. We may rewrite $\vee(a, \bot, b) \rightarrow \vee(a, b)$.

3. In a shape description, if $e \mapsto e_r$, then it should be the case that e_r is a representative of its class, i.e. $e_r \mapsto e_r$. Further, we require that subchains are contained within the map as well, i.e. if c`->`f is in the map then c is as well.

4. The list of disequalities should only mention representatives of an equivalence class, i.e. if $e_i \neq e_k$ then it should be the case that $e_i \mapsto e_i$ and $e_j \mapsto e_j$.

5. A shape description should be closed under congruence closure and not represent the bottom context. We will explain what this means in Section 4.2.1.

6. One shape description should not imply another description when both are elements of a disjunction ($\vee$). If a implies b, then we can write $a = b \wedge c$. Thus if we have $\vee(a, b)$, we can simplify $\vee(a, b) = \vee(b \wedge c, b \wedge \text{true}) = b \wedge (c \vee \text{true}) = b$. Essentially, in a disjunction between a and b, if we must prove some property about nullity in both a and b, then we will need to prove it in b. But then we would be able to prove it in a as well, without using the extra information c. Thus we can remove the extra information c without affecting the analysis. This simplification is critical to avoid the number of disjuncts from growing too large.

These invariants mean that a subcontext c is either bottom, a shape description, or a disjunction of shape descriptions. Under this representation, it is trivial to test a context for bottom, which is needed for checking assertions. Further, computing a least upper bound is also trivial: we just join the two lists using $\vee$ and simplify until we satisfy the invariants. We need to support assignment, assumption, and havocking, which are more complicated. For each of these three, we can define the operation first for operating over a shape description, and then extend it to $\vee$ forms by simply applying the operation to each shape context.

4.2 Complex Operations on Shape Descriptions

The first operation we will consider is $c \wedge e$, which is used to handle assumptions and assertions. Because we track only very limited information about the program, we only need to consider some expression forms. Further, e is a boolean expression in GCL, so the forms we have to consider are limited. We don't track information about boolean variables, so variables, dereferences, and array accesses are all ignored. We write this as $c \wedge v = c$, $c \wedge \texttt{*}p = c$ and $c \wedge a\texttt{[}i\texttt{]} = c$. We can learn from logical AND and OR expressions in GCL: $c \wedge (a \texttt{\&\&} b) = (c \wedge a) \wedge b$ and $c \wedge (a \texttt{||} b) = \vee(c \wedge a, c \wedge b)$. The ternary can also be handled this way: $c \wedge (e \texttt{?} a \texttt{:} b) = \vee(c \wedge e \wedge a, c \wedge \texttt{!} e \wedge b)$.

Comparisons between pointers are the core of what we are considering. We handle $c \wedge (p \texttt{ == } q)$ and $c \wedge (p \texttt{ != } q)$ by creating a shape description containing only the fact $p \texttt{ == } q$ or $p \texttt{ != } q$. Then we take the greatest lower bound ($\wedge$) of these two shape contexts, which requires extending the $\wedge$ operator to a pair of shape descriptions. If the comparison involves expressions which are not chains, then we cannot learn anything from it.

Expressions can also be pure calls, so the analysis will need to calculate $c \wedge f(e_1, ...)$. We calculate the new context by applying the function summary for f. To apply this summary, we need to generate a new set of variables, one for each formal parameter. Then we add to the context the fact that each variable is equal to the corresponding argument at the call site. Finally, we rename the variables `\result == true` summary, mapping each formal to its corresponding newly generated variable. We then take the greatest lower bound ($\wedge$) of this renamed summary context and the context with the variables equal to the arguments. For example if f is:

```
bool f(node p) { return p != NULL && p->f != NULL; }
```

then the function summary for `\result == true` would be

$$\text{Shape}[p \neq \texttt{NULL}, p\texttt{->}f \neq \texttt{NULL}]$$

We would process $\text{Shape}[q \neq p] \wedge f(q)$ by generating a local t for the argument, and then producing $\text{Shape}[q \neq p] \wedge t = q \wedge \text{Shape}[t \neq \texttt{NULL}, t\texttt{->}f \neq \texttt{NULL}]$. If we had not introduced t and renamed the summary, then we would have asserted that $q = p$, and introduced an incorrect contradiction.

Considering that summaries are contexts themselves, we must extend the $\wedge$ operator one step further, from expressions and shape descriptions to contexts on the right side. Thus in general, we have $c_1 \wedge c_2$, where c_1 and c_2 could each be any of $\perp$, Shape[...] or $\vee$(...). Clearly $\perp \wedge c$ and $c \wedge \perp$ are both $\perp$. We will describe shortly how to handle Shape[...]$\wedge$Shape[...], which basically means taking the facts from both. The last case is when one or both contexts are $\vee$(...). In this case, we take all of the pairwise $\wedge$ of shape descriptions from one side and the other. This has a complexity of nm, where n and m are the number of shape descriptions in each argument. This is quadratic, and causes problems for performance of the analysis. Fortunately, most of the time one or the other context is a single shape description, so the size does not always grow quickly. In addition, if any of these conjunctions ($\wedge$) produces $\perp$, then it is removed and does not contribute to the size of the resulting context.

4.2.1 Greatest Lower Bound of Shape Descriptions

Because a shape description is just a conjunction of facts about equalities and disequalities, to find the greatest lower bound we can simply take both facts together. We do this by adding the facts from the second into the first one at a time. We form a list of pairwise equalities from the second context by asserting $a = r$ whenever $a \mapsto r$ in the representative map. To add the equality, we first add the chains a and r, including any subchains which do not already exist. Then for each pair, we find the representative for both. If it is the same, then these two elements are already in the same equivalence class. Otherwise, one representative is chosen to point to the other.[2]

After this is complete, the context contains all of the equalities, but it does not necessarily have all of the consequences of these equalities. For example, if we add $a = b$ to a context in which we know $a\texttt{->}f \neq b\texttt{->}f$, then we would join the equivalence classes of a and b, but $a\texttt{->}f$ and $b\texttt{->}f$ would still be in distinct equivalence classes. This means that the context would not detect the contradiction and it would not become the bottom context.

Therefore, we run a simple version of *congruence closure* [6] to fill in these consequences. At its core, this algorithm looks for two chains $e_1\texttt{->}f$ and $e_2\texttt{->}f$ which are not in the same equivalence class, but for which the subchains e_1 and e_2 are in the same equivalence class. This can be done efficiently by grouping all chains by the equivalence class of their subchains, and then grouping by field.[3] The we can join the equivalence classes of every pair of each of these groups of elements with the same field and subchain, if they are not already in the same class. This is potentially quadratic work to check and add each equality.

After we have run closure, we can add in the inequalities from the other context. We find the representatives of each equivalence class for each inequality. If they are the same class, then we have a contradiction. Otherwise, we de-duplicate the list of inequalities and this is our context. Because we run closure and check for a contradiction, we satisfy the fifth invariant given in Section 4.1.

[2]Unlike in union find, we do this arbitrarily rather than by rank.
[3]This is done easily with a sort.

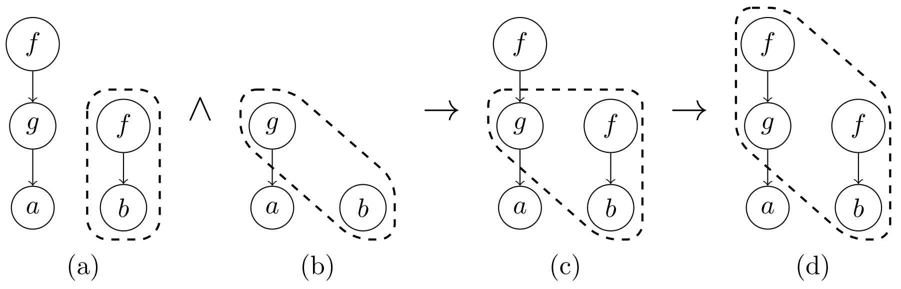


Figure 4.3: The greatest lower bound of two contexts (a) and (b). The intermediate step, before congruence closure, is shown in (c), and the final context is (d).

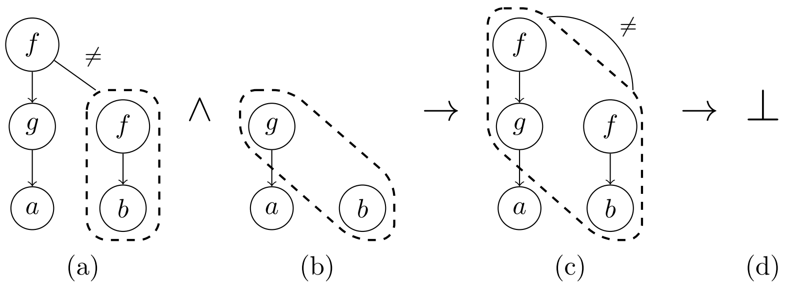


Figure 4.4: A contradicition as a result of congruence closure. The greatest lower bound of two contexts (a) and (b) results in (c) after congruence closure, which has a disequality from a class to itself. This ultimately results in $\perp$, (d).

Examples of greatest lower bounds are depicted in Figure 4.3 and Figure 4.4. In the first example, we have a context (a) in which b points to a struct whose field f points to itself. This means b and b`->`f are in the same equivalence class. The other context (b) represents the fact that a`->`$g = b$. The first context also has the chain a`->`g`->`f, but does not contain any facts about it. When the two contexts are joined, the equivalence classes for a`->`g and b are merged because they share a class in (b), which yields (c). Congruence closure finds that a`->`g and b are in the same class, and both have a child chain with the same field, but that these two fields are not in the same class (in (c), a`->`g`->`f is in a class by itself). It therefore merges these two classes, which yields (d). This was the only addition required, so the context is complete.

In Figure 4.4, the contexts are almost the same except that (a) also contains $b \neq a$`->`g`->`f. The equality processing proceeds exactly as in the previous example. However, now the disequality has both endpoints in the same equivalence class. This is a contradiction, so the result is the bottom context. If the congruence closure algorithm did not add the implied equalities, then the contradiction would not be apparent.

4.2.2 Assignment

The general form we use to analyze *lhs* `:=` *rhs* is to first compute the effects and a possible chain from the right hand side, and then make the assignment to the left hand side. Right hand sides are either an expression, an allocation or a function call. These cases are handled as follows:

1. Expression. In this case, there are no effects to consider. If the expression is syntactically a chain, then we use this as the value to store. If it is not a chain, then we store an unknown value.

2. Allocate. For this right hand side we create a fresh local variable. We add to the context that it is not `NULL`, and all fields of it (of pointer type) are `NULL`. We then make an attempt to assume in the context that this new pointer cannot be the same as any other. We look for chains already in the context of the same type. Then we add the facts that this new variable is not equal to any of these chains. This is does not always add all the facts that are true, but it suffices for most cases. The value we store is this fresh local variable.

3. Array allocate. Because we do not track arrays or their allocation status, this results in change to the context, and we store an unknown value.

4. Function call. A function call is handled by the generic analysis; the context is only given assignments of expressions or allocations.

Once we have the effect of the right hand side sorted out, we can perform the assignment itself. This also varies based on the form of the left hand side:

1. Variable, v. The simplest form is assignment to a local variable. We handle this case by first havocking that variable to remove all the old information about it, and then assuming that it is equal to the chain from the right hand side. Note that because a variable cannot appear in the right hand side if it is on the left, havocking the variable cannot destroy what we know about the right hand side. Statements such as `p = p->next` in `C0` are translated to `t := p->next; p := t` in GCL to avoid this problem. If we did not enforce this constraint on variables used on the right hand side, then our attempts to havoc `p` would destroy any nullity information we knew about `p->next`.

2. Field, p`->`f. To analyze an assignment to a field, we havoc the field and then assume that it is the value we assigned to it. However, in this case we have to be careful to only havoc fields which may alias p`->`f. In particular, if $q \neq p$, then q`->`f does not need to be havocked. Therefore during the havoc operation, we can use the disequalities to prevent information about these chains from being thrown away.

3. Cell, $*p$. As the use of cells is not present in our dataset, we ignore stores to cells by simply passing on the context unchanged. The translation will have inserted an assertion stating that p is not null, so we do not introduce an unsoundness by doing this. Instead, this ignoring of stores means that code such as:

   ```
   int** cell = alloc(int*);
   *cell = alloc(int);
   **cell = 5;
   ```

   will give an error on the third line because the analysis does not know that `*cell` is non-`NULL`.[4]

4. Array, a`[`i`]`. Because we do not track arrays, this case is trivial and handled the same way as cells.

4.2.3 Havoc

The havoc operation models unknown states via nondeterminism. It is used to update a context to perform an assignment to a variable or field and to represent the indeterminate effects of a function call or loop. There are two forms of havocking that are used: the first havocs variables and specific fields anywhere in the heap, which is used for function calls and while loops. The second havocs only fields which could be aliased to a given field, which is used for heap assignments.

[4]A simple extension is to encode cells as one field structs with deferences replaced with accesses to the field `$contents`. This would require no changes to the rest of the system and would make this example pass.

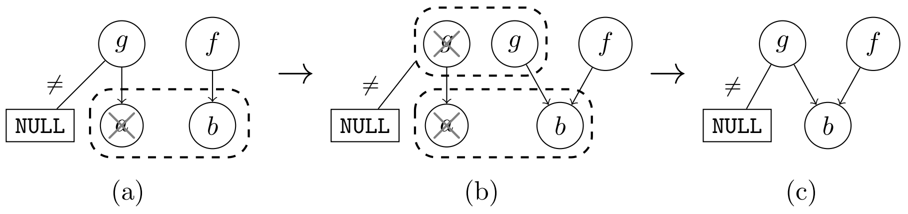


Figure 4.5: Havocking the variable a, starting with the context (a). The chain a->g must be rewritten as b->g, as shown in (b), as otherwise the nullity information about it would be lost in the final context (c).

Both cases differ in which elements of the context to erase, but once the right chains are selected, they operate identically. The basic idea is that we first find a way to represent the facts we know without referring to anything that will be thrown away by the havoc, and then we remove the havocked elements and anything which depends on them. If something cannot be written without the havocked elements, then it is something which will not be known in the new context. For example, the context Shape[a->f = a->g] expresses that the struct pointed to by a has two equal fields f and g. If we havoc a, then we are not destroying the relationship between the two fields; they are still equal. However, we do lose our way of refering to that struct. After havocking a, this context becomes Shape[], i.e. the top context or the context which has no information. If nothing else points to the a struct, all we can know is that there is *some* struct in memory with two fields aliased. It does not make sense to keep this information around, because we would never be able to make use of it.

In order to avoid losing information, we add new chains to the context which do not mention any of the elements which will be removed. In Figure 4.5, we show the process of havocking a, as would occur when it is written to in a loop. We make a distinction between the chains which themselves will be removed, and ones which are descendants of chains to be havocked. In this case, a->g is not going to be havocked, but it does depend on a. We say that a chain *will remain* if it will not be havocked and none of its sub-chains will be havocked either. In the example, both b and b->f will remain. If a chain will not remain but also will not be havocked, then it is a candidate for rewriting. In our example, this is just the chain a->g.

We can rewrite a chain by finding a subchain which has an equivalent which will remain. First we find for each equivalence class, some member which will remain, if one exists. Then, we find elements needing rewriting. We look at each subchain in turn for any which has a surviving equivalent, and add a new chain to the context if we find one. In the example, $a = b$, and so we can rewrite a->g as b->g.

After the new chains are added, we run congruence closure to ensure all equivalences are consistent, which results in (b). Here we also show that a->g will not remain, even though it is not being havocked. Then we switch the representative for each class to something which will remain, if such an element is available. Finally, we remove the elements which will not remain. This results in (c), where now all four chains are in equivalence classes by themselves. Because removed elements cannot be representatives unless their entire class is removed, we can simply remove them without breaking any of the invariants of our data structure. Chains which could not be rewritten are removed; information about them is lost. In this case, we were able to preserve information about the non-nullity of a`->`g by writing it in an alternate form.

Chapter 5

Empirical Results

In order to validate the analysis that we designed, we analyzed the student homework submissions for Principles of Imperative Computation. We selected two assignments which emphasized pointer manipulation, and which would therefore represent the kinds of programs we are targeting. The first, known as "claclab," is the first assignment where students are manipulating a pointer based data structures, and requires the students to handle linked lists, stacks, and queues. Students make a variety of elementary mistakes in this lab because pointers are new to them.

The second assignment, "editorlab," has students program the core data structure for a text editor. This program has both arrays of characters (which do not affect the pointer structure) and a doubly linked list to tie together the individual buffers. The invariants for editorlab are very involved, with many different invariant functions. Editorlab comes after claclab in the course, and students are more familiar with pointers in this assignment.

5.1 Criteria

In order to determine how well our tool might help students, we are interested in the false positive rate: the percentage of total errors which are flagged by the tool which are not actually problems in the code. If there are too many of these false positives, then students will spend too much time trying to appease or understand the tool rather than actually completing the assignment. A primary goal of our tool is to not add any additional burdens for the students. However, an error is not simply right or wrong. An error might come about for several reasons, some of which are more or less desirable. We establish four categories:

1. Actual errors. These errors are the result of bugs in student code. This can result from incorrect code or specifications which are inconsistent with the code.

2. Missing specifications. Errors of this type result from a student missing a

loop invariant, precondition, or other specification. Because our analysis is local, these properties must be stated explicitly.

3. Complex specifications. This category is for errors which are actually prevented by the specifications, but the specification (usually a precondition) is written in a way in which the tool cannot see the necessary property. This occurs most often in editorlab, where critical shape information about a field `B->point` is checked inside a loop without a loop invariant. Thus the tool does not know this information after the loop, because it can only preserve information from the invariants. If these specifications were rewritten, possibly introducing some redundancy, then the error would be resolved.

4. Tool limitations. These errors are true false positives, i.e. the specifications locally ensure correctness, but the tool is not able to correctly understand this reason, and thus produces an undesirable error message. Some errors in this category arise because the analysis sometimes attempts to prove a specification which is actually outside its domain. For example, in claclab a common postcondition is that looking up a variable after insertion should produce the inserted value, which is written `dict_lookup(D, name) == value`. Because `value` happens to have pointer type, the analysis treats this as a comparison that it should try to prove. This property is far more complicated than our analysis can handle, and so the tool gives an error for this postcondition. Depending on exactly how this is stated in the code, the tool may or may not attempt to prove it, and so an error message as actually generated only in rare cases for this particular postcondition.

5.2 Results

Figure 5.1 summarizes the results from running the analysis on a semester's worth of submissions for both claclab and editorlab. We manually classified 40 errors from each assignment, randomly selected from the errors produced by the tool. We ran our analysis only on the final submission by a student recorded on the server, which is the code that was graded. Because these are the final submissions, they are likely not to contain bugs in the code itself, but they might be still be missing specifications.

Claclab contained 434 such programs that compiled cleanly. The analysis produced a total of 166 errors. There were two programs per submission. The smaller program is an implementation of a singly linked association list. The larger program is the implementation of a calculator with programmable definitions, which manipulates a number of data structures, including stacks of integers and queues, as well as queues and the association list from the first program. The course-provided implementations of stacks and queues contain a significant number of functions, which the analysis spent a considerable amount of time analyzing. The larger program mostly manipulated data structures

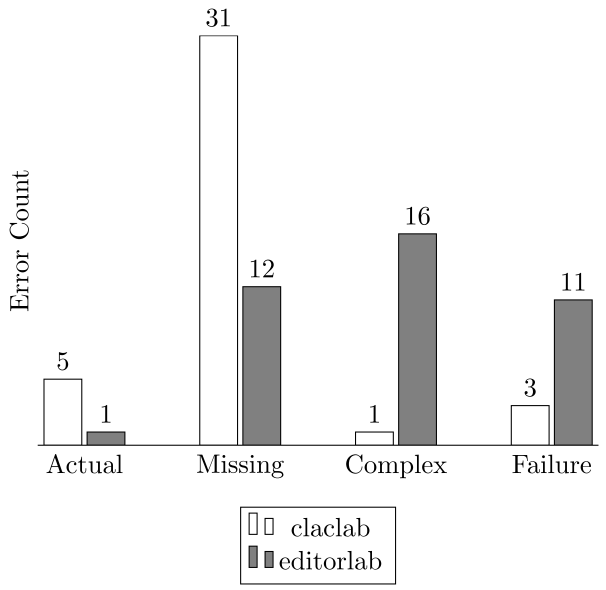


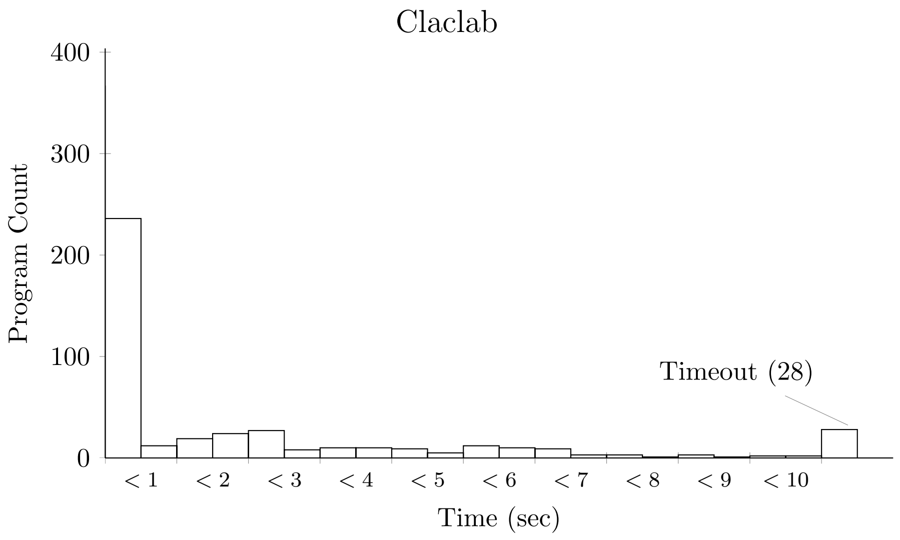
Figure 5.1: Classification of errors on editorlab and claclab. Each had 40 classified errors.

through functions, so there are few places in this program where the analysis needed to reason about pointer nullity. Most errors were thus in student implementations of association lists in the first program.

Editorlab contained a total of 739 programs. For editorlab, there were a huge number of errors reported, at 6712. There were three programs analyzed in this assignment. The first primarily manipulates arrays of characters in gap buffers which hold the text data, and so there were almost no errors in this code. The second is a implementation of text buffers, which is a doubly linked list of gap buffers. This buffer is represented with a header struct:

```
typedef struct text_buffer * tbuf;
struct text_buffer
{
  dll start;/* first node  */
  dll point;/* the cursor node */
  dll end;  /* last node */
};
```

which points to dll nodes:



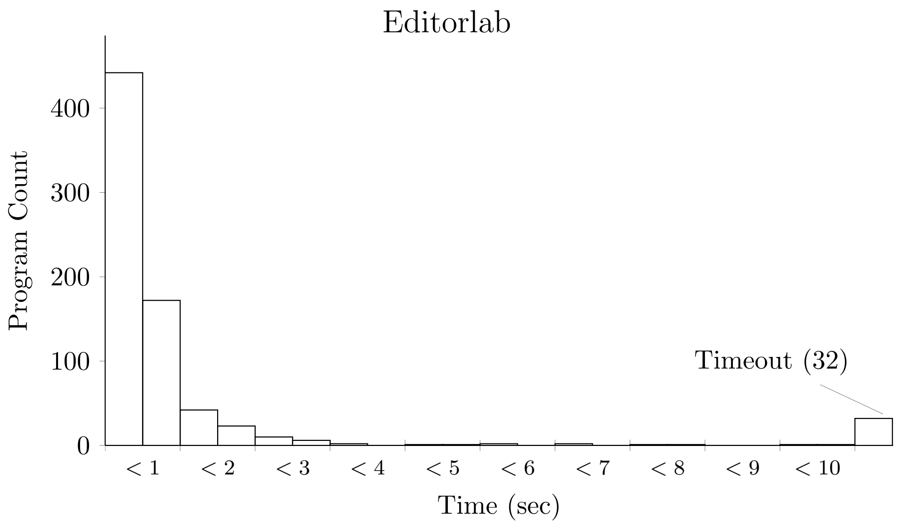


Figure 5.2: Histograms of analysis times for both labs.

```
typedef struct list_node * dll;
struct list_node
{
  elem data; /* elem == gap buffer */
  dll next;
  dll prev;
};
```

The third program ties the previous two together, by linking the invariants of the gap buffers to where they are in the list. It is in this program that most errors are generated, because it makes use of several complex invariants.

The analysis times are given in Figure 5.2. For claclab, 28 programs (6.5%) took more than 10 seconds to anlyze. Editorlab generally took less time to analyze, and 32 (4.9%) programs timed out.

5.3 Discussion

The first observation about the data in Figure 5.1 is that the errors were classified significantly further to the right categories in editorlab as compared to claclab. The classes are ordered by desirability: the best case is when a real bug is found in student code. If we discover a missing specification, then we still want it to be written explicitly, even if it is obvious to a programmer. If the tool does not understand a complex specification, then it is reasonable to require the student to state the specification more directly, but ideally the tool can use the code as written. Finally, we would like to minimize cases when the analysis produces an entirely unwarranted error due to its own limitations.

Claclab had most errors in the first two categories, with a vast majority classified as missing specifications. There were some real bugs caught by the tool, and only a few failures. In contrast, editorlab was dominated by errors in the final two categories, with missing specifications still remaining frequent. In general we can say that the quality of errors was much higher on claclab than on editorlab.

The data in Figure 5.2 indicates that there were a small but non-trivial number of programs for which the tool did not complete in our interactive response window of ten seconds. As a fraction of the total programs, there were more timeouts on claclab, and the histogram show that there is an increase in the number of programs finishing in a given time until about 3 seconds. Despite claclab having overall simpler code, there was a significantly higher volume of it to analyze.

The second claclab program uses two implementations of stacks and one of queues, which is a significant number of functions to analyze. Because C0 lacks the ability to write generic, polymorphic data structures, there were two largely identical implementations of stacks, and the tool needed to analyze this code afresh for each program. The large number of programs analyzed in a half second are primarily the association list implementation, which was a relatively small amount of code. This variation in program size results in the bimodal distribution seen for claclab.

While some of these timeouts are caused by simple code volume, most are caused by the way redundancy is introduced by the $\vee$ operation on contexts. Because we effectively have a single overall list of disjuncts, the analysis cannot compactly represent information about separate, independent parts of the state. For example, consider the following program fragment:

```
if(p != NULL) { ... //@assert p->f != NULL; ...}
else {...}
if(q != NULL) { ... //@assert q->g != NULL; ...}
else {...}
```

The context after the first if is $(p \neq \texttt{NULL} \wedge p\texttt{->}f \neq \texttt{NULL}) \vee (p = \texttt{NULL})$. When the second if is analyzed, the nullity of q is independent of p, but the context becomes:

$$\begin{array}{l}(p \neq \texttt{NULL} \wedge p\texttt{->}f \neq \texttt{NULL} \wedge q = \texttt{NULL}) \\ \vee(p = \texttt{NULL} \wedge q = \texttt{NULL}) \\ \vee(p \neq \texttt{NULL} \wedge p\texttt{->}f \neq \texttt{NULL} \wedge q \neq \texttt{NULL} \wedge q\texttt{->}f \neq \texttt{NULL}) \\ \vee(p = \texttt{NULL} \wedge q \neq \texttt{NULL} \wedge q\texttt{->}f \neq \texttt{NULL})\end{array}$$

Effectively the analysis has flattened the structure, "multiplying out" the two conditional contexts. In programs which branch on pointer comparisons or have many independent disjunctions, this can mean an exponential increase in the the number of state descriptions the analysis must consider. This size increase cannot be prevented by the implication simplification given in Section 4.1, because none of these descriptions imply one another. In this sense, these are disjunctions which we would like to preserve, but our representation forces the size of the context to grow substantially. If we were able to represent disjunctions in a factored form, along the lines of:

$$(p \neq \texttt{NULL} \wedge p\texttt{->}f \neq \texttt{NULL} \vee p = \texttt{NULL}) \wedge (q \neq \texttt{NULL} \wedge q\texttt{->}f \neq \texttt{NULL} \vee q = \texttt{NULL})$$

then we could represent these separate disjunctions in linear space, but this would increase the complexity of the analysis because we would need to decide when to expand and when to factor.

5.3.1 Claclab

In claclab, by far the most common error was a missing loop invariant on a while loop which traverses a linked list. They would typically be structured like this:

```
while(p->next != NULL)
{
    if (...) return;
    p = p->next;
}
```

To be fully correct, this loop requires the invariant `p != NULL`, as otherwise the loop test is not safe. This is an example of a trivial invariant which is often omitted, but which our tool would require students to write down.

An example of an actual bug found by the tool is given by this fragment:[1]

```
alist* searcher = D->assoclist;
while (searcher != NULL)
  searcher = searcher->next;
... // no assignments to searcher
searcher->next = add; // unprotected pointer dereference
```

In this example, the loop guarantees that the variable `searcher` will be null after the loop. Later, the student dereferences this variable, which will definitely crash at runtime. The correct program would need to have a loop guard `searcher->next != NULL`, so the variable itself is never null.

Missing loop invariants are not the only case where a specification is required. In this program:

```
queue Q = dict_lookup(D, name);
alist* temp = D->assoclist;
if (Q != NULL) {
  string s = temp->name; // unprotected pointer dereference
  ...
}
```

dereferencing `temp` in the if statement is safe, but the specifications to show this are missing. The critical property is that if a non-null queue `Q` is found in the dictionary `D`, then there must have been at least one entry. Therefore `D->assoclist` cannot be null, as otherwise the list would be empty. This property should be stated as a postcondition of `dict_lookup`, but in all programs which relied on this property, this postcondition was missing.

The tool failures resulted from the analysis trying to prove properties which it was not capable of reasoning about. For example, a function in one program has the postcondition `dict_lookup(D, name) != NULL || def == NULL`, which intends to express that that the value inserted by the function can really be found. The reason this is a tool failure is because this is really a correctness property, not a safety property. The tool interprets this as an assertion that it should prove rather than as a property outside its domain.

In general, the tool found real errors or missing invariants in claclab. This feedback would encourage students to write more specifications (especially loop invariants), and in a few cases actually help them identify bugs in their code.

5.3.2 Editorlab

Editorlab presented a much more complicated data structure, and the tool performed significantly worse on these programs. The primary driver of many

[1]This code and the other examples here have been re-formatted for brevity and clarity, but represent equivalent code to what the analysis considered.

complex specification errors is the following program fragment, in which `B` is a header struct which points to the first and last elements of a doubly linked list, as well as a `point` somewhere in this list:

```
bool is_linked(tbuf B) {
  dll p = B->start;
  bool has_point = false;
  while(p != B->end)
    //@loop_invariant p != NULL;
  {
    if(p == B->point) has_point = true;
    p = p->next;
  }
  return has_point;
}
  ...
  //@assert is_linked(B);
  B->point->data = ...;
```

The implicit invariant in the while loop is that if the point was found, i.e. `has_point` is true, then the point is non-null. In the full editorlab example, we also typically need some information about the `prev` and `next` fields around the point. There is an implicit invariant in the loop: `!has_point || B->point != NULL`. Without inference of invariants or a programmer supplied one, the analysis is not able to reason that if this function returns true, then the point is safe to dereference. This pattern is the reason that so many editorlab examples have "complex specification" errors. One possible solution is to have students write the conditions that the `point` must satisfy separately, which means these conditions must be redundantly checked.

In addition to complex specifications, there were also many examples of simply missing specifications as in claclab. For example, the following program is an example of a incomplete specification function:

```
bool is_linked(tbuf B)
  //@requires B != NULL;
{
    if (B->point == B->start || B->point == B->end) return false;
    return true;
}
bool tbuf_at_left(tbuf B)
  //@requires is_linked(B);
{
    // unprotected pointer dereference [B->start may be null]
    return (B->start->next == B->point);
}
```

The function `is_linked` is supposed to check that the text buffer `B` is properly linked, but only checks two properties the data structure is supposed to have. A complete function would check that `B->start` and `B->end` are non-null, which is needed for `tbuf_at_left`, as well as the fact that the data structure is a doubly linked list.

Another example involves a student not properly establishing the preconditions of a function:

```
bool is_linked(tbuf B)
  //@requires B != NULL;
  //@requires B->start != NULL;
{ ... }
bool tbuf_at_left(tbuf B)
  //@requires is_linked(B);
  //@ensures is_linked(B);
{ ... }
```

Here the preconditions of `tbuf_at_left` are not well formed because the preconditions of `is_linked` are not established before it is called. To fix this, we would need to either move these conditions out of the preconditions and into the body of the checker function (which is more idiomatic `C0`), or we would need to add the necessary preconditions to `tbuf_at_left`:

```
bool tbuf_at_left(tbuf B)
  //@requires B != NULL;
  //@requires B->start != NULL;
  //@requires is_linked(B);
  ...
```

Editorlab produced a large number of tool failures; errors for which the specifications are sufficient and the program is correct, but which the tool gave an error. These are the least desirable errors because they cause students to lose confidence in the tool or search for a non-existent bug. One example of these failures is due to the way function calls are translated outside of specifications. This is manifest in the following program:

```
  if (!is_gapbuf(last->data))
    return false;
  // unprotected pointer dereference [last->data may be null]
  //@assert (\length(last->data->buffer)==16);
```

The definition of `is_gapbuf` ensures that if it returns true, then its argument is non-null. However, the analysis still gives an error on the line after the if. The translation in GCL looks like:

```
  b = is_gapbuf(last->data);
  if (!b)
    return false;
  //@assert (\length(last->data->buffer)==16);
```

ignoring the translation of if into blocks and breaks, which just pushes the problem deeper. The problem is caused by the fact that the return value of `is_gapbuf` is captured to a variable, which is then branched on in the if. Because we do not represent information about boolean variables, the conditional information from the function's postconditions is lost when the analysis finishes processing the assignment to `b`. We cannot use the specification trick of translating to a pure call in the condition, because in general the heap may change; we need to capture to a variable to prevent this. In addition, in other instances the function that is called is not a specification function, so this strategy would not solve all cases.

Editorlab was a significant challenge for our analysis. In order to effectively use this tool on this assignment, students would likely need to be instructed how to write specifications which are more amenable to analysis. In addition, it would need to be modified to support using information from functions, specification or otherwise, which are called in conditions. Finally, the explosion in state size would need to be addressed.

Chapter 6

Conclusions

We present a static analysis tool embedded within the C0 compiler which analyzes a program and reports errors for pointer dereferences which may crash the program at runtime. This analysis first translates C0 into a specially crafted intermediate form with a regular structure. A pointer analysis was developed which represents the state of the program as one of a number of possible shapes, where each shape is described by the equalities and disequalities that hold between the pointer valued locations in the program. This information is sufficient to verify whether a pointer dereference is safe, and is easy to understand and debug because it has an immediate interpretation at the C0 source level.

We presented the results of running our analysis on a corpus of real student submissions. We found that on the early assignment, with simpler data structures and code, that the analysis was able to identify real problems with a low false positive rate. Because students need the most help early in the course, when pointers and data structures are new, this analysis will likely be sufficient and useful for at least this assignment. On more complicated programs, the inability to track information from more complex invariants, as well as the general lack of specifications written by the students, caused the number of errors to be overwhelmingly high. In order to use our tool on these assignments, either students will need to write specifications in an idiomatic style or the analysis will need to become more powerful.

Our analysis has a number of properties which make it a good fit for the larger educational context for which the tool was intended. By relying on the specifications that the course encourages students to write down, we can enable a local analysis which is both sound and does not give too many undesirable errors. Our direct representation of the heap closely mirrors how students are taught to reason about their code formally. Our analysis is fast, enabling it to be used interactively during the development of a program, allowing students to get immediate feedback on their code.

6.1 Future Work

There two primary directions in which this work could be extended: evaluation in with live students, and improving the analysis itself. While the results of analyzing our corpus indicates that our approach is potentially viable, a comparative analysis, either with random trials or via comparing across semesters, would give a definitive answer as to whether this approach helps students. Observing how students interact with the tool and how it shapes the way they program would be particularly interesting avenues to explore. This information would be helpful both in building future iterations of the tool and also with understanding the pedagogy of programming education. Given the popularity of teaching coding at a variety of levels, from K-12 through post-secondary, this could be valuable data for informing those curriculums.

The analysis tool itself can be extended in several ways. The analysis tool fails to capture certain invariants that are expressed through loops within specification functions. While some of these can be rewritten to make the critical property evident to the tool, properties such as the shape of a list are only expressible through such loops. It would be helpful, especially on later assignments, if the tool was able to infer invariants or the properties these loops are expressing. It might also be beneficial for the tool to infer trivial invariants on loops within the main code, either to not burden the programmer with writing them down or to allow them to be automatically suggested.

In addition, the tool could be extended with integer reasoning to help students with array out of bounds errors, which are problem as common as null pointer errors in the first few assignments which emphasize arrays. Our analysis currently only gives errors. A more useful system might also be able to suggest fixes or give hints to students about why their code is not correct.

Finally, the tool could be made to save its analysis between runs, so that parts of the program which do not change can have their summaries reused rather than calculated every time a program is analyzed. Properly implemented, this could reduce the time to analyze a program while it is under development to a fraction of a second even for large programs, making it lightweight enough to be run continuously during development.

References

[1] R. Arnold. "C_0, an Imperative Programming Language for Novice Computer Scientists". M.S. Thesis, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, PA, 2010.

[2] C0 Informational Homepage. `http://c0.typesafety.net/`

[3] E. W. Dijkstra. "Guarded commands, non-determinacy and formal derivation of programs." *Commun. ACM 18*, 8: 453457, 1975.

[4] D. Hovemeyer, J. Spacco, W. Pugh. "Evaluating and tuning a static analysis to find null pointer bugs." In Proc. of PASTE '05. Michael Ernst and Thomas Jensen (Eds.). ACM, New York, NY, USA, 13-19. DOI:10.1145/1108792.1108798, 2005.

[5] K. Rustan M. Leino. "This is Boogie 2." Microsoft, Redmond, WA, Manuscript KRML 178, June 2008.

[6] G. Nelson, D. C. Oppen. "Fast Decision Procedures Based on Congruence Closure." *Journal of the ACM*, Vol 27, No. 2, April 1980.

[7] A. Subramanian. "Compiler Generation for Substructural Operational Semantics." M.S. Thesis, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, PA, 2012.