CSCI-598 Program Verification and Synthesis
Final Project Proposal

David Florness

February 27th, 2020

1 Introduction

Compilers are unique software in that they grant programmers using them the freedom to write their software elegantly and clearly without said programmers needing to micro-optimize nor sacrifice clarity for the sake of optimal program execution. In other words, compilers, in addition to fulfilling their primary role of translating source code written in one language to another, can perform optimizations for the programmer.

One such optimization compilers often make is removing bits of code that allocate memory that is never used meaningfully at any point in the program. Allocating memory and never using it is a waste of both time (as it takes time to request memory from the operating system) and, of course, memory itself.

We propose a means of using program synthesis to implement this optimizer.

2 Examples

Consider the following C program.

```
int main(int argc, char *argv[])
{
        int i, a;

        for (i = 0; i < 100; ++i)
                a += i;

        return 0;
}
```

Executing the `for` loop is wasteful since it only modifies the variable `a`, whose value is never used. Thus, we would want our compiler to remove said `for` loop along with the declarations for `i` and `a`.

Unfortunately, removing code that unnecessarily allocates memory is not always as simple as removing variable references and declarations of said variables. This is illustrated in the following example.

```
#include <limits.h>
#include <stdio.h>

#define BUFLEN 4096

int main(int argc, char *argv[])
{
        char buf[BUFLEN];
        int i;

        for (i = 0; i < BUFLEN; ++i)
                buf[i] = i % CHAR_MAX;

        for (i = 0; i < BUFLEN/2; ++i)
                printf("%d\n", buf[i]);

        return 0;
}
```

In this case, we cannot remove all of the `buf` array; we can only remove its latter half. A similar scenario can occur for members of a struct:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

#define MAXLINE 1024

struct person {
        char name[MAXLINE];
        uint8_t age;
};

int main(int argc, char *argv[])
{
        struct person p;
        printf("Enter your name: \n");
        fgets(p.name, MAXLINE, stdin);
        printf("Hello, %s", p.name);
        return 0;
}
```

The `age` member of the `person` struct is never used and thus we can safely remove it from

the `person` struct definition.

3 Proposed Approach

For this project, our optimizer will only operate on programs written in the C programming language. This is because C is a simple language but also shares a common structure with many other programming languages. Many languages are categorized as "C-like" for this reason.

We will first represent a given C program as an abstract syntax tree (AST), which gives us an easy means of traversing the semantics of the program. ASTs are used by virtually all compilers to internally represent a program.

We will then use linear temporal logic (LTL) to assert that if, in the AST, we come across code that allocates memory (whether on the stack or the heap), we should see that memory used meaningfully later in the program. For a hunk of memory to be used "meaningfully", we mean that it must either be

1. used as an input to a system call (syscall), or
2. used in computing a value that is ultimately given to a syscall.

Syscalls are how programs interface with the host operating system. From a user's perspective, any variables that never play a part in affecting invocations of syscalls are meaningless.

Note: ofttimes the C programming language is used to write software for embedded devices that use global variables of the program to control certain aspects of the hardware. We do not want our optimizer to remove such declarations, even if they're never used in syscalls throughout the program. Fortunately, the C programming language provides the `volatile` keyword to indicate variables of this nature.

4 Proposed Evaluation

Ultimately, our optimizer will be "correct" if and only if it produces programs that are semantically identical to its input programs. Therefore, our means of ensuring that semantics are preserved will be to create unit tests for our input programs and ensuring that the outputs of the optimized program are identical to the original program.