Ensuring That You're Using The Memory That You Allocated

David Florness

April 23rd, 2020

The Problem

1. Programmers define variables to store values in memory.
2. A programmer may forget to delete variables that are never meaningfully use as s/he develops a given program.
3. Why don't we help out the programmer?

Example

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

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

        return 0;
}
```

A 2nd Example

```c
#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;
}
```

High-level Overview of the Solution

1. Take a C program and parse it into an AST. For parsers that don't support macros, we can pass it through the C Prepprocessor, cpp, first.
2. For every variable that we come across, make a linear temporal logic assertion that the said variable will be used meaningfully later on in the program.
3. The programmer is alerted of any variable that breaks this assumption.

Parsing a C Program

Racket Parsing Program

```
#lang racket/base
(require c)

(define program
  ; returns a list of declarations
  ; for the C program
  (parse-program (open-input-file
                  "/path/to/c/file.c")))
```

/path/to/c/file.c

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

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

        return 0;
}
```

The Abstract Syntax Tree

```
parse-c.rkt> program
'(#s((decl:function decl 1)
     #s(src 1 1 0 99 10 3 #f)
     #f
     #f
     #s((type:primitive type 1) #s(src 1 1 0 4 1 3 #f) int)
     #s((decl:declarator decl 1)
        #s(src 5 1 4 33 1 32 #f)
        #s((id:var id 1) #s(src 5 1 4 9 1 8 #f) main)
        #s((type:function type 1)
           #s(src 9 1 8 33 1 32 #f)
           #f
           (#s((decl:formal decl 1)
               #s(src 10 1 9 18 1 17 #f)
               #f
               #s((type:primitive type 1) #s(src 10 1 9 13 1 12 #f) int)
               #s((decl:declarator decl 1)
                  #s(src 14 1 13 18 1 17 #f)
                  #s((id:var id 1) #s(src 14 1 13 18 1 17 #f) argc)
                  #f
                  ...
```

Rosette

1. Racket[1] is a special LISP in that it makes it very easy to create new programming languages.
2. Rosette extends Racket with language constructs for program synthesis, verification, and more. To verify or synthesize code, Rosette compiles it to logical constraints solved with off-the-shelf SMT solvers like Z3. By combining virtualized access to solvers with Racket's metaprogramming, Rosette makes it easy to develop synthesis and verification tools for new languages.

[1] `https://racket-lang.org`

Verifying that Two Functions Are The Same

```
#lang rosette
(define (poly x)
 (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (factored x)
 (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
 (assert (= (p x) (f x))))

(same poly factored 0)
(same poly factored -1)
(same poly factored -2)
```

Verifying that Two Functions Are The Same

```
#lang rosette
(define (poly x)
 (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (factored x)
 (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
 (assert (= (p x) (f x))))

(define-symbolic i integer?)
(define cex (verify (same poly factored i))) ; (model [i 14])
(println (poly 14))       ; 57120
(println (factored 14))   ; 53760
```

AST to LTL

1. Using the parse tree, we can create Racket functions that behave exactly like the original C functions but return vectors of booleans indicating whether the variables were used. Inputs of the function can be replaced by Rosette symbolics. Other non-constant unknowns can also be replaced by Rosette symbolics.
2. Symbolics can be booleans, integers, reals, bitvectors, and uninterpreted functions.

   ```
   (define-symbolic i integer?)
   ```

3. For each variable, we can then just ask Rosette to find at least one case where that variable is used.