| galois |

Lab: Memory Safety

Recall the C function `saf` is defined like this in `saf.c`:

```
uint8_t* saf (uint8_t *a, uint8_t b[10]) {
   uint8_t i,x;
   if (*a < 0 || *a > 9) *a = 0;
   else *a = b[*a];
   return a;
}
```

and the Cryptol function `saf` is defined like this in `saf.cry`:

```
saf : [1][8] -> [10][8] -> [1][8]
saf a b = if a@0 > 9 then [0]
          else if a@0 < 0 then [0]
          else [(b@(a@0))]
```

Consider the small change where instead of a pointer output there is just a `uint8_t` type in `saf.c` and instead of a sequence output there is just a `[8]` type output in `saf.cry`. The saw file must change accordingly. The current `saf.saw` file produces output with

```
llvm_points_to pidx (llvm_term {{ saf idx arr }});
```

But now the return value is not a pointer so just the following is sufficient instead:

```
llvm_return (llvm_term {{ saf idx arr }});
```

Exercise 1:

The C function `saf` is changed to this:

```
uint8_t saf (uint8_t *a, uint8_t b[10]) {
   uint8_t x;
   if (*a < 0 || *a > 9) x = 0;
   else x = b[*a];
   return x;
}
```

The Cryptol `saf` function is changed to this for compatibility:

```
saf : [1][8] -> [10][8] -> [8]
saf a b = if a@0 > 9 then 0
          else if a@0 < 0 then 0
          else (b@(a@0))
```

Change `saf.saw` to prove that the C function is safe and equivalent to the Cryptol function. Be sure to use `:safe` on the Cryptol function and use `clang` to generate the llvm bitcode for the C function. ■

Consider another small change where, instead of `uint8_t *a` as the first argument to the C function `saf`, there is just `uint8_t a` and instead of the signature of the Cryptol `saf` function being:

```
saf : [1][8] -> [10][8] -> [8]
```

it becomes:

```
saf : [8] -> [10][8] -> [8]
```

Then instead of the following lines in `saf.saw`:

```
idx <- llvm_fresh_var "index" (llvm_array 1 (llvm_int 8));
pidx <- llvm_alloc (llvm_array 1 (llvm_int 8));
llvm_points_to pidx (llvm_term idx);
```

which are needed when the first argument of `saf` is a pointer, the following line becomes a replacement for them:

```
idx <- llvm_fresh_var "index" (llvm_int 8);
```

Exercise 2:

The C function `saf` is changed to this:

```
uint8_t saf (uint8_t a, uint8_t b[10]) {
   uint8_t x;
   if (a < 0 || a > 9) x = 0;
   else x = b[a];
   return x;
}
```

The Cryptol `saf` function is changed to this for compatibility:

```
saf : [8] -> [10][8] -> [8]
saf a b = if a > 9 then 0
          else if a < 0 then 0
          else (b@a)
```

Change `saf.saw` to prove that the C function is safe and equivalent to the Cryptol function. Be sure to use `:safe` on the Cryptol function and use `clang` to generate the llvm bitcode for the C function. ■

Cryptol file `zero.cry` contains the following two functions:

```
usaf : [8] -> [8] -> [8]
usaf a b = (a/b):[8]

saf : [8] -> [8] -> [8]
saf a b = if b == 0 then 1 else (a/b)
```

Function `saf` has protection against a divide by 0 but `usaf` does not. Consider the C function `saf` in file `zero.c`:

```
uint8_t saf (uint8_t x, uint8_t y) {
   if (y == 0) return 1;
   else return x/y;
}
```

and function `usaf` in file `uzero.c`:

```
uint8_t usaf (uint8_t x, uint8_t y) {
   return x/y;
}
```

Exercise 3:

Apply `:safe` to both functions in `zero.cry`. What is the result? Create a saw file named `zero.saw` that verifies the C function `saf` is equivalent to the Cryptol function `saf`. Run saw `zero.saw` and display the result. Create a file named `uzero.saw` that shows the C function `usaf` cannot be verified safe against the Cryptol `saf` function. Run `saw uzero.saw` and display the result. ■

The following three C functions are equivalent and return a count of the '1' bits in a word:

```
/* From Henry S. Warren Jr.'s Hacker's Delight */
int pop_count(uint32_t x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

```
/* A version of popcount that uses multiplication */
int pop_count_mul(uint32_t x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = ((x + (x >> 4)) & 0x0F0F0F0F);
   return (x * 0x01010101) >> 24;
}
```

```
/* A version of popcount that uses an indefinite while loop(!) */
int pop_count_sparse(uint32_t x) {
  int n;
   n = 0;
   while (x != 0) {
      n = n + 1;
      x = x & (x - 1);
   }
   return n;
}
```

Exercise 4:

Put the above functions in a C file named `popcount.c`.
Run clang to get the llvm bitcode of these functions in file `popcount.bc`.
Write a Cryptol function that does what the above functions do in `popcount.cry`.
Show that this function is safe,
Write a SAW file `popcount.saw` that proves equivalence of all functions, C and Cryptol.
Run the SAW file.