| galois |

Lab: Functional Correctness

Consider this file, `point.c`:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

typedef struct point { uint32_t x; uint32_t y; } point;

point ZERO = {0, 0};

// Check whether two points are equal
bool point_eq(const point *p1, const point *p2) {
   return p1->x == p2->x && p1->y == p2-> y;
}

// Allocate and return a new point
point* point_new(uint32_t x, uint32_t y) {
   point* ret = malloc(sizeof(point));
   ret->x = x;
   ret->y = y;
   return ret;
}

// Return a new point containing a copy of `p`
point* point_copy(const point* p) {  return point_new(p->x, p->y); }

// Add two points
point* point_add(const point *p1, const point *p2) {
   // Save an addition by checking for zero
   if (point_eq(p1, &ZERO)) return point_copy(p2);
   if (point_eq(p2, &ZERO)) return point_copy(p1);
   return point_new(p1->x + p2->x, p1->y + p2->y);
}

int main (int argc, char **argv) {
   point *p1, *p2, *p3;
   p1 = point_new(11,34);
   p2 = point_new(5,12);
   p3 = point_new(23,16);
   printf("p1=%d,%d p2=%d,%d p3=%d,%d\n",p1->x,p1->y,p2->x,p2->y,p3->x,p3->y);
   p3 = point_add(p1,p2);
   printf("p1=%d,%d p2=%d,%d p3=%d,%d\n",p1->x,p1->y,p2->x,p2->y,p3->x,p3->y);
   p2 = point_copy(p3);
   printf("p1=%d,%d p2=%d,%d p3=%d,%d\n",p1->x,p1->y,p2->x,p2->y,p3->x,p3->y);
   printf("p2 and p3 equal? %d\n",point_eq(p2,p3));
   printf("p1 and p3 equal? %d\n",point_eq(p1,p3));
}
```

Exercise 1:
Show that C function `point_eq(p1,p2)`, where `p1` and `p2` are pointers to `point struct`, is equivalent to the Cryptol specification `[p1' == p2']` where `p1'` and `p2'` are the values that `p1` and `p2` are pointing to. That is, create SAW file `s1.saw` and run `saw` on it.

Here are some hints. The following which, given a variable `name`, creates the variable of that name and returns a pair consisting of a pointer to the space allocated to the named `llvm` variable and the named `llvm` variable:

```
let fresh_point_readonly name = do {
   p_ptr <- llvm_alloc_readonly (llvm_struct "struct.point");
   p_x <- llvm_fresh_var (str_concat name ".x") (llvm_int 32);
   p_y <- llvm_fresh_var (str_concat name ".y") (llvm_int 32);
   llvm_points_to p_ptr (llvm_struct_value [ llvm_term p_x, llvm_term p_y]);
   let p = {{ { x = p_x, y = p_y } }};
   return (p, p_ptr);
};
```

Then creating two variables, say `p1` and `p2`, with pointers is easy like this:

```
(p1, p1_ptr) <- fresh_point_readonly "p1";
(p2, p2_ptr) <- fresh_point_readonly "p2";
```

The `point_eq` function operates on the pointers like this:

```
llvm_execute_func [p1_ptr, p2_ptr];
```

The return value should be equivalent to the Cryptol `[p1 == p2]` (no pointers in Cryptol but that is OK because we have `p1` and `p2` that are not). The reason for the brackets is that SAW doesn't yet support translating Cryptol's bit type(s) into crucible-llvm's type system. ■

Exercise 2:
Prove `point_new` returns a `struct point` object equivalent to what a Cryptol type `Point` would build with the same inputs (see `Point.cry`). Make and add Cryptol Point objects like this (See Point.cry):

```
Point> let pt1 = {x=5,y=34}
Point> let pt2 = {x=7,y=16}
Point> let pt3 = point_add pt1 pt2
Point> pt3
{x = 12, y = 50}
```

The following SAW function, which creates space for a `Point` object `p` and returns a pointer to that space, may be useful in solving this problem.

```
let alloc_assign_point p = do {
   p_ptr <- llvm_alloc (llvm_struct "struct.point");
   llvm_points_to p_ptr (llvm_struct_value
                          [ llvm_term {{ p.x }}, llvm_term {{ p.y }}]);
   return p_ptr;
};
```

It can be used like this in `point_new_setup` where `p_x` and `p_y` are `fresh_llvm` variables that represent the `x` and `y` numbers in a `struct point` object and a `Point` object.

```
ret_ptr <- alloc_assign_point {{ {x = p_x, y = p_y } }};
```

Observe the correspondence between the C `struct point` and the Cryptol `Point`. ■

Exercise 3:
Prove `point_copy` returns a new `struct point` object that is equivalent to an input `point` object. The `point_copy` setup in SAW will create a `(p,p_ptr)` pair using `fresh_point_readonly` (from above), the C function `point_copy` will take `p_ptr` as input, and `alloc_assign_point` (from above) will use a pointer to `p`, a Cryptol `Point`, as return value. ■

Exercise 4:
Prove `point_add`, with input of two `point` objects, creates a new `point` object whose `x` component is the sum of `x` components of the input `point` objects and whose `y` component is the sum of `y` components of the input `point` objects. To save an addition the point_add function returns a copy of one point if the other point is zero. In all three cases the addition is the same. A Cryptol function is written as a specification in `Point.cry` as follows:

```
point_add : Point -> Point -> Point
point_add p1 p2 = { x = p1.x + p2.x, y = p1.y + p2.y }
```

The SAW file will '`import "Point.cry"`'. SAW function `alloc_assign_point` will be used as above to return a Cryptol value, this time from the Cryptol `point_add` above. As before, pairs `(p1,p1_ptr)` and `(p2,p2_ptr)` will be created using `fresh_point_readonly` as above. The `p1` and `p2` will be used by the Cryptol `point_add` and the `p1_ptr`, `p2_ptr` are used as arguments to `llvm_execute_func`. What's different this time is that the global variable ZERO but be taken into account in the setup function. This can be done with

```
let zero_term = llvm_term {{ 0 : [32] }};
llvm_alloc_global "ZERO";
llvm_points_to (llvm_global "ZERO")
               (llvm_struct_value [zero_term, zero_term]); ■
```