- Every value is associated with an location. A location can be a variable x, an abstract location variable ϕ (which represent a set of locations) or a part of another location (such as a field inside a record). The formal definition of location is as follows:

$$\begin{aligned} Field &\ni l \\ Location \ni L &::= \ x \mid \phi \mid L.l. \end{aligned}$$

- There can be subset constraints between locations. For example $\phi_1 \supset \phi_2$ indicate that the location set represented by ϕ_1 must be a superset of that of ϕ_2.

- Every value has an *access* to its location, which can be `own`, `imm` or `mut`, representing ownership, immutable and mutable borrow in respect.

- Similar to Rust, a location can have *either* many immutable borrows *or* a unique mutable borrow at the same time (exclusively). Also, on assignment or ownership movement to a location, no borrow can exist.

- Unlike Rust, the above constraints is not guaranteed by using a lifetime system. Instead, the above constraints is guaranteed by two steps:

  1. At every program point, the location system provide approximation of alias information. Using this information, we can determine whether a value has certain access to some locations.
  2. At every borrow/assignment/ownership movement, we make use of the alias information and *kill* all values whose existence will violate the constraints.

- Consider the following example program:

```
let x = ref 1;
let y = &x;
x := 2;
print_int !y
```

  Here x (trivially) has ownership access to the location x, while y has immutable borrow access to x. Now on the third line of the program, since we are assigning to x, we *kill* all values that has borrow access to x: including y. Now on the fourth line, since y is already killed, using it triggers a type error.

- The first use of location variables is to make the type-based alias analysis more accurate. Consider the following example:

```
let x = ref 1;
let y = ref 2;
let borrow_of_x = &x;
let borrow_of_xy = if (some_condition) { &x } else { &y };
y := 3;
print_int !borrow_of_x; // should be OK
print_int !borrow_of_xy; // should fail
```

To assign a type to `borrow_of_xy`, we must equate the types of two branches of the `if`-expression. But the first branch has type `&x ref(int)`, while the second branch has type `&y ref(int)` (Assume we annotate location of a borrow right after the ampersand). So we assign the type `&L ref(int)` to `borrow_of_xy`, where `L` is a fresh location variable, and introduce the subset constraints $L \supset x$ and $L \supset y$. Now when assigning to `y`, all borrows of `y` are killed. Since $L \supset y$, all borrows of `L` must be killed, too, for soundness. Hence `borrow_of_xy` is killed, while `borrow_of_x` is not.

- The novelity of the system lies in its capability to deal with higher order functions. Higher order functions are hard to deal with because it is very difficult to "abstract over constraints". But with the location sysetm, there is a simple way to eliminate all constraints while preserving soundness: lift all subset constraints to equality constraints (substitution).
- As an example, consider the following example:

```
fn f(r : &mut ref(&int), v : &int) {
    r := v;
}
```

Its most precise type is (omitting unnecessary annotations):

```
f : (r : &mut ref(&L1 int), v : &L2 int) -> ()
    where L2 is a subset of L1
```

A less precise, but constraint-free type is:

```
f : (r : &mut ref(&L int), v : &L int) -> ()
```

since trivially, $L \supset L$.