Path

A path is a sequence of one or more path segments separated by a namespace qualifier (::). If a path consists of only one segment, it refers to either anitem or avariable in the scope. If a path has multiple segments, it always refers to an item. Two examples of paths consisting of only identifier segments: x; x::y::z; There are two kinds of paths: generic and concrete.

Generic paths

A generic path is composed of segments which are identifiers, e.g.x::y::z . It refers to an item. Generic paths are used at: * Use statements. * Enum variant pattern.

Concrete paths

Concrete paths are composed of concrete segments, which are identifiers that may be followed by ageneric argument clause like this: IDENT[[::]] The additional:: is mandatory where ambiguity would otherwise arise, such as in expressions. For example,x::::y::::z is a concrete path. Otherwise, it can be omitted. For example, in type expressionsfn foo(x: Option) . A concrete path refers to a concrete item (e.g. a function, type, etc.). When the generic arguments clause is not provided, or less generic arguments are provided than the item expects, the compilerinfers the missing generic arguments. Examples: fn foo(a: A, b: B) { ... }

fn main() { foo::(false, 3_u2); // OK foo::(false, 3_u2); // OK, B is inferred to be u32 foo::(false, 3_u2); // OK, B is inferred to be u32 foo(false, 3_u2); // OK, A, B are inferred as bool and u32 respectively } Concrete paths are used at: * Expressions foo::bar::() // Function call expression Option::::Some(true) // Enum variant expression * Type expressions * * let x: Option= None; // Let statement type clause * * Option::> // Generic type argument * Trait expressions impl MyImpl of MyTrait { // trait expression after of keyword ... } fn foo>() { ... } // trait expression after: in an impl generic argument

Path resolution

The path is resolved by looking up the first segment identifier in the current local scope. This includes local variables and generic parameters defined in containing items, up to the module level. If not found, it is looked up in the core module. Note that it is not looked up in the containing modules. If the found item has a scope (e.g. a module or an enum) then the second identifier is looked up inside this scope, and so on.

4.9 Traits and impls 5.2 Namespaces ð§