Generics

Generic programming is a way to define "recipes" for creating concrete items . This recipe has parameters called "generic parameters" (in addition to the "normal" parameters). For example: fn foo(x: T) -> T { x } This is a recipe for creating an identity function for any typeT . Items such asfunctions , structs, enums, traits, impls, extern types and type aliases can be generic. They are defined using a comma-separated list, enclosed by angle brackets (<…>), immediately following the item's name. If an item doesn't require generic parameters, the whole<…> clause can be omitted. Each generic parameter has a name that is used to refer to it in the item's body. For example, the generic structBox below defines a type that can hold any type of data: fn main() { struct Box { value: T, }

```
let box_u32 = Box::<u32> { value: 5_u32 };
let box_u128 = Box::<u128> { value: 1_u128 };
let box_box_u32 = Box::<Box::<u32>> { value: 100_u32 };
```

} In this example,T is a generic parameter. When substituted with concrete types/data, the "recipe" forms a concrete item. Multiple concrete items can be formed from the same "recipe". The concrete values that substitute the generic parameters are called generic arguments. In the above example,u32 andu128 are generic arguments. In the last statement,u32 is a generic argument to create the concrete typeBox:: (aBox that holdsu32) andBox:: is then used as a generic argument to create another concrete type:Box::> aBox that holds aBox that holdsu32 .

Types of Generic Parameters

Cairo supports several types of generic parameters:

Type Generic Parameters

Type generic parameters are used to define generic types. This is the default type of a generic parameter, thus to add such a generic parameter, you should simply specify a name for it. For example, the generic structBox above has a type generic parameter with the nameT .

Impl Generic Parameters

Impl generic parameters are used to declare impls that exist in the scope of the item. It can be used by the item when it refers to the relevant trait. This allows the item to use trait functions, while allowing its users to determine which impl will be actually used. For example: trait MulTrait { fn mul(x: T) -> T; } impl Double of MulTrait:: { fn mul(x: u32) -> u32 { x * 2 } } impl Triple of MulTrait:: { fn mul(x: u32) -> u32 { x * 3 } } fn foo>(x: T) -> T { MyImpl::mul(x) } fn main() { let x = foo::>(5); assert(x == 10, 'Should be doubled');

```
let y = foo::<u32, Triple::<u32>>(5);
assert(y == 15, 'Should be tripled');
```

} The generic functionfoo defines an impl generic parameter namedMyImpl .main uses 2 concrete forms offoo . One uses theDouble implementation of theMulTrait trait, and one uses theTriple implementation of theMulTrait trait. Even if the impl generic parameter name is not used in the item body, the fact that it was specified as a generic parameter can be used to apply some restrictions to the generic types used by the item. For example,foo from the above example can't have a concrete form in whichT isu128 as there is no impl ofMulTrait foru128 . In this case, you can also use anonymous impl generic parameters: fn foo>(x: T) -> T { MyTrait::mul(x) } Note that in many cases generic arguments can be inferred and thus can be omitted from the use of the item.

Const Generic Parameters

Const generic parameters are used to define generic constants. It is currently very partially supported and should not be used.

Generic arguments

Generic arguments are the concrete values that are used to instantiate generic parameters. They are specified inpaths that describe generic items. Each kind of generic parameter has its own kind of generic arguments: * Type generic parameters are instantiated with type expressions. * Impl generic parameters are instantiated with impl expressions. * Const generic parameters are instantiated with const expressions. These are not supported yet.

Named Generic Arguments

Generic arguments can be named. This allows specifying them in any order, and also allows specifying only some of them. For example, the following two statements are equivalent: let x = foo::(5); let x = foo::(5);

9.2 Linear Types 9.4 Inference