Linear types

Linear types are a way to statically ensure that a value is used exactly once. Cairo supports linear types by havingmove semantics and forbidding copying and dropping values by default.

Move semantics

When passing a value to a function, it is moved by default. This means that after a value is used for the first time, it cannot be used again. For example, the following code does not compile: struct A {}

fn main() { let a = A {}; foo(a); // value is passed by value once here. foo(a); // error: Value was previously moved. } To allow a value to be used multiple times, theCopy trait must be implemented for it. For example, the following code compiles:

[derive(Copy)]

struct A {}

fn main() { let a = A {}; foo(a); // value is passed by value once here. foo(a); // Now there is no error. } Note: the snapshot operator@ (See the docs about snapshot) is considered not moving a value.

Clone

Sometime a value can’t be trivially copied, but we can still use code to build a copy of it. For example, a value containing anArray cannot be copied, but we can still clone it by cloning the array it contains with.clone() . This can be done by implementing or deriving theClone trait. The derived implementation requires that all fields implementClone , and will automatically call.clone() on all the fields.

Variable dropping

By default, a value may not go out of scope unless it was previously moved. For example, the following code does not compile:

[derive(Copy)]

struct A {}

fn main() { A {}; // error: Value not dropped. } To allow a value to be dropped, theDrop trait must be implemented for it. For example, the following code compiles:

[derive(Drop)]

struct A {}

fn main() { A {}; // Now there is no error. }

Destructors

Sometime a value must not be dropped, but we can still use code to get rid of it. For example, a value containing aDict cannot be dropped, but we can still deconstruct it by destructing the dict it contains with.destruct() . This can be done by implementing or deriving theDestruct trait, which will be called automatically when a non-droppable value goes out of scope. The derived implementation requires that all fields implementDestruct , and will automatically call.destruct() on all the fields.

[derive(Destruct)]

struct A { d: Dict }

fn main() { A {}; // No error, A will be destructed. } When implementingDestruct manually, note that the implementation must benopanic , because destructors are called when a value goes out of scope, which may happen in a panic.

Copy and drop restrictions

Copy cannot be implemented for a type that contains a non-copyable field. Similarly,Drop cannot be implemented for a type that contains a non-droppable field. Some basic data types of Cairo are inherently non-copyable and non-droppable.Array is not copyable, whileDict is not copyable nor droppable. The reason for this has to do with Cairo’s immutable memory model.

Snapshot

The snapshot type is always copyable and droppable. It is used to create an immutable snapshot of a value.

Common pitfalls and solutions

- How to avoid "Value was previously moved" errors?
  - Use@
  - to create a snapshot of the value.
  - Useref
  - to pass the value by reference.
  - Implement or deriveCopy
  - to allow the value to be copied.
  - Implement or deriveClone
  - to allow the value to be cloned.
  - For a generic parameter, add another generic parameter for Copy or Clone (e.g.impl TCopy: Copy
  -).
- How to avoid "Value was not dropped" errors?
  - Implement or deriveDrop
  - to allow the value to be dropped.
  - Implement or deriveDestruct
  - to allow the value to be destructed.
  - For structs, deconstruct them usinglet A { .. } = a;
  - .
  - For enums, deconstruct them usingmatch
  - .
  - In particular, for the 'never' type, match like this:match x {}
  - .
  - For a generic parameter, add another generic parameter for Drop or Destruct (e.g.impl TDrop: Drop
  -).
  - Find a function that can be used to destroy the value, and call it.

9.1.11 Dict type 9.3 Generics