Part of the Carbon Language, under the Apache License v2.0 with LLVM Exceptions.
SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception

Carbon Language - http://github.com/carbon-language

Multi-Thread Examples

Authors: danakj, ...
Status: Draft
Created: 2025-10-24
Docs stored in Carbon's Shared Drive Access instructions

Problem

Examples that demonstrate parallelism and concurrency needs for Carbon. Either code that should compile, or problematic code that should not.

Examples

Thread spawn

- No join, so no way to know when it's done in the caller.
- Passing in a pointer that is invalidated

```
None
fn F() {
  var i: i32 = 0;
  var p: const i32* = &i;
  spawn(fn [p] => G(p));
  // Bad: i is destroyed, p is invalidated but in use on another thread.
}
```

- Holding a pointer that is invalidated, moving its owner

```
None
fn F() {
  var i: Box(i32) = .Make(0);
  var p: const i32* = i.Get();
  spawn(fn [~i] => ~i));
```

```
  // Bad: i may be destroyed, then p is invalidated.
  Core.Print(*p);
}
```

- Holding a pointer that is invalidated, not moving its owner

None

```
var i: Cell(Optional(i32)): .Make(.Some(0));

fn F() {
  var p: const i32* = i.Get();
  spawn(fn => i.Set(.None()));
  // Bad: The i32 in i may be destroyed/inactive, then p is invalidated.
  Core.Print(*p);
}
```

- Long lived vs short lived pointers

None

```
fn F(pointer_to_string_literal: const char*, pointer_to_string: const char*) {
  spawn(fn [let] => H(pointer_to_string_literal, pointer_to_string);
}

fn G1() {
  let world = "world" as strbuf;
  F("hello", world.Get());
  // Bad: `world` is freed, but the pointer is in use on another thread.
}

fn G2() {
  F("hello", "world");
  // OK: neither pointer is freed while in use on another thread.
}
```

- Long lived with mutation

None

```
fn F(pointer_to_global: const i32*) {
  spawn(fn [let] => H(pointer_to_global);
}

var g: i32 = 2;

fn G() {
  let world = "world" as strbuf;
  F(&g);
  // Bad: Writing to `g: i32` on this thread, but maybe reading it on another
thread.
  g = 3;
}
```

- Passing unowned along with ownership

None

```
fn F() {
  var b: Box(i32) = .Make(2);
  var p: i32* = b.Get();
  spawn(fn [~b, p] => H(~b, p));
  // Bad: `p` needs to be invalidated for soundness. The box may be destroyed
  // at any time.
  Core.Print(*p);
}
```

- Passing combined owned, unowned, and global/static, pointer to locally-owned, and Rc.

None

```
class C(^B, ^G, ^L) {
  // Can move across threads.
  var b: Box(i32);
  // Moves with the owner, is valid. Must invalidate any copies on the original
thread.
  var p: i32 ^B *;
  // Global so can move across threads. Must not invalidate any copies on the
original
  // thread.
```

```
  var g: i32 ^G *;
  // Moves across threads without the other, not valid. Must be an error.
  var l: i32 ^L *;
  // Owns its data, but can not move across threads. Must be an error.
  var r: Rc(i32);
}

var globalint: i32 = 3;

fn F() {
  var b: Box(i32) = .Make(2);
  var p: i32* = &c.b;
  var g: i32* = &globalint;
  var l: i32 = 1;
  var r: Rc(i32) = .Make(4);
  var c: C = { .b = ~b, .p = p, .g = g, .l = l, .r = r };
  // Two errors must occur:
  // - Sending `.l` across threads without its owner. The owner will invalidate
  //   it in the future.
  // - Sending `.r` across threads, as its _type_ is not valid to move across
  //   threads even though it owns its data.
  spawn(fn [~c] => H(~c));
  // Error: `p` needs to be invalidated for soundness. The box may be destroyed
  // at any time.
  Core.Print(*p);
  // Accepted: `g` must *not* be invalidated, as it points to a global not an
  // owned thing.
  Core.Print(*g);
}
```

Send effect

Maybe we need a `send` effect/function capability. Like we can say `invalidate(x)` we could say `send(x)` to say we move it (or a copy maybe?) to another thread. Then we can say no based on conditions like:

- `Rc` is **not** allowed (how do we mark it as such? Type capabilities?)
- `.l` is allowed by type but then we can look to see there's **not** an owner going with? And see there is not.
- `.b` is allowed by type and we can see that there **is** an owner going with.
- `send(x)` would invalidate any alias sets owned by something in `x`. So `p` gets invalidated, but `g` doesn't.

```
None
fn spawn[Func: Call](func: Func) [[send(^func.any)]];
```

- Problem is that anything that gets `send` applied to it needs to pass that up the call stack to anything that passed it along. Rust only requires putting `Send` on generics that pass things through a call stack, since types get `Send` as an auto trait.

```
None
class C;

fn GenericSpawns[T:! type](c: T) [[send(c)]];
fn SpawnsGenerically(c: C) [[send(c)]];
fn CallsSpawnsGenerically(c: C) [[send(c)]];
```

Vs Rust

```
None
struct C;

fn GenericSpawns<T: Send>(c: T);
fn SpawnsGenerically(c: C);
fn CallsSpawnsGenerically(c: C);
```

Scoped thread spawn

- Mostly for the ability to pass in pointers to data in the caller, knowing when the thread is done with them.
    - To allow **immutable** sharing
    - Not to allow destruction from the thread
- A fixed point in the caller where the task is done running.
- Either scoped as a lambda: gives a nested scope for the point where the thread ends
- Or as a return guard that *must be destroyed*: allows returning the guard, giving a multi-function/non-local scope where the thread may be running

- spawn_scoped allows pointers to things before the scope

```
None
fn F() {
  var i: i32 = 0;
```

```
  var p: const i32* = &i;
  var scope: auto = spawn_scoped(fn [p] => G(p));
  ~scope;
  // Scope does not finish destruction until the scoped_spawn is joined:
  // i was not destroyed and p was not invalidated.
}
```

- But not if they are mutated in the scope.

```
None
fn F() {
  var i: Optional(i32) = 0;
  var p: const i32* = &i.Get();
  var scope: auto = spawn_scoped(fn [p] => G(p));
  // Bad, the i32 in i is gone, so p is invalidated but it's in use on another
thread.
  i = .None;
  ~scope;
}
```

- And doesn't allow holding pointers to things given to the thread

```
None
fn F() {
  var i: Box(i32) = .Make(0);
  var p: const i32* = i.Get();
  var scope: auto = spawn_scoped(fn [~i] => ~i);
  // Still bad: i may be destroyed, so p may be invalidated.
  Core.Print(*p);
  ~scope;
}
```

Coroutines

- Web server calling backends

```
None
class Request {
  var data: buf(u8),
}
class Headers {
  var mode: Optional(i32);
```

```
  var pos: slice(u8);  // In Request.data.
}

/*async*/ fn ParseHeaders(let r: Request) -> Future(Headers);
/*async*/ fn QueryDatabase(let h: Headers) -> Future(strbuf);
/*async*/ fn SendReply(reply: strbuf) -> Future(());

/*async*/ fn HandleRequest(var r: Request) -> Future(()) {
  let headers: Headers = ParseHeaders(r).await;
  var answer: strbuf = QueryDatabase(headers).await;
  return SendReply(answer);  // returns the Promise/Future instead of .await.
}

/*async*/ fn ClearsDataDuringHandleRequest(var r: Request) -> Future(()) {
  let headers: Headers = ParseHeaders(r).await;
  // Bad: Invalidates headers.pos, but headers is used below.
  r.data.clear();
  var answer: strbuf = QueryDatabase(headers).await;
  return SendReply(answer);  // returns the Promise/Future instead of .await.
}

/*async*/ fn RequestDoesntLiveLongEnough(let r: Request*) -> Future(()) {
  let headers: Headers = ParseHeaders(*r).await;
  // Bad: r could be invalidated as soon as we call .await, while ParseHeaders()
  // is working if it has any .await inside.
  return SendReply("done");  // returns the Promise/Future instead of .await.
}
```

Fiber race

(adapted from a Rust example)

None

```
// Run each of the supplied children concurrently on its own
fiber
// that is a child of the parent making this call. Return the
result
// of first child to finish, cancelling and joining the
```

```
// others.
//
// In Rust:
// pub fn race<'a, T>(children: Vec<Box<dyn 'a + FnOnce() -> T +
Send>>) -> T

fn race[T:! Type](children: Vec<Box<call!() -> T>> -> T {
    // There is no correct answer if the set of children is
empty.
    assert!(!children.is_empty());

    // Spawn child fibers that race to store their result into
    // a OnceLock. After one of them wins, cancel and join them
all.
    // OnceLock is mutable, but shared in Rust
    // (using interior mutability)
    let once = OnceLock::Make();
    nursery::with_nursery([ref once](n: Nursery) {
        // Start children competing to be the first.
        let fibers: Vec<Fiber> = children
            .map(|child| {
                // Run child in a a fiber
                n.spawn(|| {
                    let _ = once.set(child());
                })
            })
            .collect();

        // Wait for one to win.
        once.wait();

        // We've got what we need. Clean up.
        for (f in fibers) {
            f.cancel();
            f.join();
        }
```

```
    });

    // By now our result is in the OnceLock.
    Box::Take(once.consume())
}
```