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

Safety Unit No. 10: private unsafe / public safe

Authors: josh11b, (add yourself)
Status: Draft
Created: 2025-10-14
Docs stored in Carbon's Shared Drive Access instructions

"Safety Unit" is a series of docs with "units of discussion": a doc to help written, async communication.

Proposal

We don't expect to be able to make all data structures have safe implementations, but we want them to have safe APIs. To accomplish that, we want to limit the code that needs to be verified as maintaining the invariants needed for safety. The idea is to strengthen the public/private split already used to hide implementation details, and then only code with private access (members of the class and friends) needs to be reviewed.

Lets see how it could work using an example:

```
None
class StackListWithDefault(T:! type) {
  private class Node;
  private var ^A: Node;

  class Node {
    var data: T;
    var next: Optional(Node ^A *);
  }

  private var top: Optional(Node ^A *);
  // Having a default value makes the empty case simpler.
  private var deflt: T;

  // public, type of contained places is `T`.
  alias ^Elts = ^A.data + &deflt;

  fn Make(deflt: T) -> Self;
```

```
  fn Push[ref self: Self](x: T) [[write(^self)]] {
    // TODO: should there be an effect to indicate that a
    // newly allocated place is added to `^A`?
    // ...
  }
  fn Pop[ref self: Self]() -> T [[write(^self), free(^Elts)]] {
    if let .Some(p: Node ^A *) = self.top {
      returned var r: T = p->data;
      self.top = p->next;
      // UNSAFE: actually invalidating ^A, but we only have
      // `[[free(^Elts)]]` effect
      Allocator.Release(p);
      return var;
    } else {
      return self.deflt;
    }
  }

  fn Peek[ref self: Self]() -> ref ^Elts T {
    if let .Some(p: Node ^A *) = self.top {
      return p->data;
    } else {
      return self.deflt;
    }
  }

  // TODO: destructor
}
```

Another example, in sketch, is the `RC` class. It will also publicly expose access (including references) to a contained data item of type `T`. It would like to prevent public access to the reference count itself. Furthermore, `RC`'s destructor will invalidate references to `T` data elements, but not the pointers in other `RC` instances (even though that isn't safe).

Goals:

- Easier to invalidate the public slice of an alias set even when suppressing invalidation of a private alias set containing it.
- Provide a way to reacquire references to elements in a container after they have been invalidated by a shape-changing operation.
- Provide a guarantee that only code with private access has a pointer to any object in an owned private alias set (as long as unsafe code doesn't leak it).

Something that makes this tricky is that we are assuming unsafe code, for which we are generally not restricting or providing guarantees for. However, by switching from Rust's unsafe blocks to unsafe *operations*, we hopefully can say that some unsafe operations preserve these properties, and are expressive enough for common data structures. Other unsafe operations would require greater scrutiny.

Discussion on 2025-10-06
#safety 2025-10-03