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

Safety Unit No. 14: interface effects

Authors: josh11b, (add yourself)
Status: Draft
Created: 2025-10-25
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.

Example

in safety unit no. 11, we identified that instances where type information is erased need particular care.

```
Java
interface A {
  // Move allows For Swap().
  let T:! Core.Move;
  fn B[ref self: Self]() -> ref T;
  fn C[ref self: Self]() -> ref T;
}

interface D {
  // QUESTION: What effects can we put here, given that we can't
  // talk about the fields of `Self`?
  fn F[ref self: Self]();
}

impl forall [U:! A] U as D {
  fn F[ref self: Self]() {
    Swap(ref self.B(), ref self.C());
  }
}

// QUESTION: What are the effects of `G`?
fn G[V:! A](ref x: V) {
  x.(D.F)();
}
```

```
class E {
  // pointers to be returned by B and C
  var b: i32**;
  var c: i32**;
  // other pointers to be swapped by a specialization of `D`
  var p: i32**;

  impl as A where .T = i32** {
    fn B[ref self: Self]() -> ref T { return self.b; }
    fn C[ref self: Self]() -> ref T { return self.c; }
  }
  // specialization overriding the blanket impl of D in terms of A
  impl as D {
    fn F[ref self: Self]()
        // Can have specific effects here. Note: getting pretty verbose!
        // Longer than the function itself.
        [[^out.*self.c = ^*self.p, ^out.*self.p = ^*self.c,
          write(...)]] {
      Swap(ref *self.c, ref *self.p);
    }
  }
}

fn H() {
  var e: E = {...};
  // QUESTION: What are the effects of this call?
  G(ref e);
}
```

There are a few big concerns:

- Difficulty of specifying effects on interfaces, since we can't reference specifics of the type.
- Effects for specific implementations can vary widely, won't be accurately captured by effect annotations on the interface.
- Effects may come from a different interface than those mentioned in the function's constraints due to blanket `impl`s.
- Specialization adds further uncertainty about what an interface function can or does do.
- Some operations, like swap, affect individual alias sets without affecting the union we would use to summarize when you don't have the individual fields.

Idea: delegated effects

Observe that the specific results of a copy operation are different for different types. We want to preserve which specific fields have captured which references. A specific type might implement "copy" in a way that doesn't just copy. So we have some unspecific effects on the interface that are used to check uses within a generic, but the concrete consumer of the generic would like to see the more specific effects from the impl of the interface for the specific type. So instead of including the net effect from the interface in the effect signature, the idea is to mention the interface itself and let the caller compute the effects with whatever more specific information the caller has.

QUESTION: How would that work?

Rejected option: effects in associated constants

One possibility is for an interface to have an *associated effect constant* (or just "associated effect" analogous to "associated type") as in

```
None
interface Destroy {
  let E:! Core.Effect;
  fn Op[ref self: Self]() [[with(E)]]
}
```

But then we have a problem of decoupling effects from the parameters and then later rebinding them, as discussed on 2025-11-05.

Option: `unknown` and `call` effects

Better instead to get rid of `Core.Effect` values entirely, and use the delegated effects idea both for interface methods and for function types:

```
None
fn G[FN:! call[ref](ref C) [[unknown]]
    ](ref h: FN) [[ calls(h(...)) ]];

interface Destroy {
  fn Op[ref self: Self]() [[unknown]];
}
```

```
fn F2[T:! Destroy](ref a: T, ref b: T) [[ calls(a.(Destroy.op)()) ]]
```

This would imply a `call(...)` effect for delegated effects. This requires a good default broad effect for `unknown` (maybe similar to the defaults for unannotated C++ functions?) that can be used to check generically, or an explicit broad effect specified in the interface for type checking generics that all `impl`s of that interface would be constrained by.

Option: `template` effects

Since the definitions of generic functions are visible to callers, we can delay checking of effects until after they are instantiated/monomorphized.

Advantages:

- Easier onramp to memory safety by avoiding the need to write down long and tricky function contracts.
- Precise effects based on the impls used for concrete types.

Disadvantages

- The usual problems with templates:
  - They can fail based on new uses.
  - Poor experience when there are compile failures: diagnostics can only say what happened, not what contract was violated nor whether the problem is in the caller or callee.
  - They expose the implementation as part of the function's contract, making evolution difficult.

To avoid `template` being viral to all callers, it can be contained by an assertion that a set of effects are bounded by an explicit effects expression.

My expectation is that we will have `template` effects but minimize their use, in line with our generics strategy.

Option: Effect constraints in facet types

The idea here is that a generic function using an interface as a constraint has more specific knowledge of what effects it can allow and still pass effect checking. So if the generic function could write `T:! MyInterface where`... and then specify effect constraints on the methods of `MyInterface` that `T`'s `impl` of `MyInterface` would have to satisfy. Two big concerns:

- The effect constraints would need to be expressed in terms of the parameters to those methods, which are not currently in scope.
- This would generally be quite verbose.

In Rust

https://rust.godbolt.org/z/xsY545Er4

Rust

```rust
trait Write {
    fn write(&mut self);
}

trait A {
  type T : Write;
  fn B(&mut self) -> &mut Self::T;
  fn C(&mut self) -> &mut Self::T;
}

trait D {
  fn F(&mut self);
}

impl<U: A> D for U {
  fn F(&mut self) {
    self.B().write();
  }
}

// QUESTION: What are the effects of `G`?
fn G<V: A>(x: &mut V) {
  <V as D>::F(x);
}

struct E<'e> {
  // pointers to be returned by B and C
  b: &'e mut &'e mut i32,
  c: &'e mut &'e mut i32,
  // other pointers to be swapped by a specialization of `D`
  p: &'e mut &'e mut i32,
}
impl<'e> A for E<'e> {
  type T = &'e mut &'e mut i32;
  fn B(&mut self) -> &mut Self::T { &mut self.b }
  fn C(&mut self) -> &mut Self::T { &mut self.c }
}

impl Write for &'_ mut &'_ mut i32 {
  fn write(&mut self) {
    ***self *= 2;
```

```
  }
}

// specialization overriding the blanket impl of D in terms of A
/*
impl as D {
  fn F[ref self: Self]() {
      Swap(ref *self.c, ref *self.p);
  }
}
*/

fn H() {
  let mut i1 = 0;
  let mut p1 = &mut i1;
  let mut i2 = 0;
  let mut p2 = &mut i2;
  let mut i3 = 0;
  let mut p3 = &mut i3;

  let mut e = E{
    b: &mut p1,
    c: &mut p2,
    p: &mut p3,
  };
  G(&mut e);
}
```