Inner items do not inherit type or const parameters from the functions they are embedded in.

Erroneous code example:

```
fn foo<T>(x: T) {
    fn bar(y: T) { // T is defined in the "outer" function
        // ..
    }
    bar(x);
}
```

Nor will this:

```
fn foo<T>(x: T) {
    type MaybeT = Option<T>;
    // ...
}
```

Or this:

```
fn foo<T>(x: T) {
    struct Foo {
        x: T,
    }
    // ...
}
```

Items inside functions are basically just like top-level items, except that they can only be used from the function they are in.

There are a couple of solutions for this.

If the item is a function, you may use a closure:

```
fn foo<T>(x: T) {
    let bar = |y: T| { // explicit type annotation may not be necessary
        // ..
    };
    bar(x);
}
```

For a generic item, you can copy over the parameters:

```
fn foo<T>(x: T) {
    fn bar<T>(y: T) {
        // ..
    }
    bar(x);
}
```

```
fn foo<T>(x: T) {
    type MaybeT<T> = Option<T>;
}
```

Be sure to copy over any bounds as well:

```
fn foo<T: Copy>(x: T) {
    fn bar<T: Copy>(y: T) {
        // ..
    }
    bar(x);
}
```

```
fn foo<T: Copy>(x: T) {
    struct Foo<T: Copy> {
        x: T,
    }
}
```

This may require additional type hints in the function body.

In case the item is a function inside an `impl`, defining a private helper function might be easier:

```
# struct Foo<T>(T);
impl<T> Foo<T> {
    pub fn foo(&self, x: T) {
        self.bar(x);
    }

    fn bar(&self, y: T) {
        // ..
    }
}
```

For default impls in traits, the private helper solution won't work, however closures or copying the parameters should still work.