An unboxed trait object was used as a return value.

Erroneous code example:

```
trait T {
    fn bar(&self);
}
struct S(usize);
impl T for S {
    fn bar(&self) {}
}

// Having the trait `T` as return type is invalid because
// unboxed trait objects do not have a statically known size:
fn foo() -> dyn T { // error!
    S(42)
}
```

Return types cannot be `dyn Trait`s as they must be `Sized`.

To avoid the error there are a couple of options.

If there is a single type involved, you can use `impl Trait`:

```
# trait T {
#     fn bar(&self);
# }
# struct S(usize);
# impl T for S {
#     fn bar(&self) {}
# }
// The compiler will select `S(usize)` as the materialized return type of this
// function, but callers will only know that the return type implements `T`.
fn foo() -> impl T { // ok!
    S(42)
}
```

If there are multiple types involved, the only way you care to interact with them is through the trait's interface, and having to rely on dynamic dispatch is acceptable, then you can use trait objects with `Box`, or other container types like `Rc` or `Arc`:

```
# trait T {
#     fn bar(&self);
# }
# struct S(usize);
# impl T for S {
#     fn bar(&self) {}
# }
struct O(&'static str);
```

```
impl T for O {
    fn bar(&self) {}
}

// This now returns a "trait object" and callers are only be able to access
// associated items from `T`.
fn foo(x: bool) -> Box<dyn T> { // ok!
    if x {
        Box::new(S(42))
    } else {
        Box::new(O("val"))
    }
}
```

Finally, if you wish to still be able to access the original type, you can create a new `enum` with a variant for each type:

```
# trait T {
#     fn bar(&self);
# }
# struct S(usize);
# impl T for S {
#     fn bar(&self) {}
# }
# struct O(&'static str);
# impl T for O {
#     fn bar(&self) {}
# }
enum E {
    S(S),
    O(O),
}

// The caller can access the original types directly, but it needs to match on
// the returned `enum E`.
fn foo(x: bool) -> E {
    if x {
        E::S(S(42))
    } else {
        E::O(O("val"))
    }
}
```

You can even implement the `trait` on the returned `enum` so the callers *don't* have to match on the returned value to invoke the associated items:

```
# trait T {
#     fn bar(&self);
```

```
# }
# struct S(usize);
# impl T for S {
#     fn bar(&self) {}
# }
# struct O(&'static str);
# impl T for O {
#     fn bar(&self) {}
# }
# enum E {
#     S(S),
#     O(O),
# }
impl T for E {
    fn bar(&self) {
        match self {
            E::S(s) => s.bar(),
            E::O(o) => o.bar(),
        }
    }
}
```

If you decide to use trait objects, be aware that these rely on dynamic dispatch, which has performance implications, as the compiler needs to emit code that will figure out which method to call *at runtime* instead of during compilation. Using trait objects we are trading flexibility for performance.