A type parameter that is specified for `impl` is not constrained.

Erroneous code example:

```
struct Foo;

impl<T: Default> Foo {
    // error: the type parameter `T` is not constrained by the impl trait, self
    // type, or predicates [E0207]
    fn get(&self) -> T {
        <T as Default>::default()
    }
}
```

Any type parameter of an `impl` must meet at least one of the following criteria:

- it appears in the *implementing type* of the impl, e.g. `impl<T> Foo<T>`
- for a trait impl, it appears in the *implemented trait*, e.g. `impl<T> SomeTrait<T> for Foo`
- it is bound as an associated type, e.g. `impl<T, U> SomeTrait for T where T: AnotherTrait<AssocType=U>`

Error example 1

Suppose we have a struct `Foo` and we would like to define some methods for it. The previous code example has a definition which leads to a compiler error:

The problem is that the parameter `T` does not appear in the implementing type (`Foo`) of the impl. In this case, we can fix the error by moving the type parameter from the `impl` to the method `get` :

```
struct Foo;

// Move the type parameter from the impl to the method
impl Foo {
    fn get<T: Default>(&self) -> T {
        <T as Default>::default()
    }
}
```

Error example 2

As another example, suppose we have a `Maker` trait and want to establish a type `FooMaker` that makes `Foo` s:

```
trait Maker {
    type Item;
    fn make(&mut self) -> Self::Item;
}

struct Foo<T> {
    foo: T
}

struct FooMaker;
```

```
impl<T: Default> Maker for FooMaker {
// error: the type parameter `T` is not constrained by the impl trait, self
// type, or predicates [E0207]
    type Item = Foo<T>;

    fn make(&mut self) -> Foo<T> {
        Foo { foo: <T as Default>::default() }
    }
}
```

This fails to compile because `T` does not appear in the trait or in the implementing type.

One way to work around this is to introduce a phantom type parameter into `FooMaker`, like so:

```
use std::marker::PhantomData;

trait Maker {
    type Item;
    fn make(&mut self) -> Self::Item;
}

struct Foo<T> {
    foo: T
}

// Add a type parameter to `FooMaker`
struct FooMaker<T> {
    phantom: PhantomData<T>,
}

impl<T: Default> Maker for FooMaker<T> {
    type Item = Foo<T>;

    fn make(&mut self) -> Foo<T> {
        Foo {
            foo: <T as Default>::default(),
        }
    }
}
```

Another way is to do away with the associated type in `Maker` and use an input type parameter instead:

```
// Use a type parameter instead of an associated type here
trait Maker<Item> {
    fn make(&mut self) -> Item;
}

struct Foo<T> {
    foo: T
}
```

```
struct FooMaker;

impl<T: Default> Maker<Foo<T>> for FooMaker {
    fn make(&mut self) -> Foo<T> {
        Foo { foo: <T as Default>::default() }
    }
}
```

Additional information

For more information, please see RFC 447.