- Start Date: 2014-06-10
- RFC PR: rust-lang/rfcs#48
- Rust Issue: rust-lang/rust#5527

Summary

Cleanup the trait, method, and operator semantics so that they are well-defined and cover more use cases. A high-level summary of the changes is as follows:

1. Generalize explicit self types beyond `&self` and `&mut self` etc, so that self-type declarations like `self: Rc<Self>` become possible.
2. Expand coherence rules to operate recursively and distinguish orphans more carefully.
3. Revise vtable resolution algorithm to be gradual.
4. Revise method resolution algorithm in terms of vtable resolution.

This RFC excludes discussion of associated types and multidimensional type classes, which will be the subject of a follow-up RFC.

Motivation

The current trait system is ill-specified and inadequate. Its implementation dates from a rather different language. It should be put onto a surer footing.

Use cases

Poor interaction with overloadable deref and index

Addressed by: New method resolution algorithm.

The deref operator `*` is a flexible one. Imagine a pointer `p` of type `~T`. This same `*` operator can be used for three distinct purposes, depending on context.

1. Create an immutable referent to the referent: `&*p`.
2. Create a mutable reference to the referent: `&mut *p`.
3. Copy/move the contents of the referent: `consume(*p)`.

Not all of these operations are supported by all types. In fact, because most smart pointers represent aliasable data, they will only support the creation of immutable references (e.g., `Rc`, `Gc`). Other smart pointers (e.g., the `RefMut` type returned by `RefCell`) support mutable or immutable references, but not moves. Finally, a type that owns its data (like, indeed, `~T`) might support #3.

To reflect this, we use distinct traits for the various operators. (In fact, we don't currently have a trait for copying/moving the contents, this could be a distinct RFC (ed., I'm still thinking this over myself, there are non-trivial interactions)).

Unfortunately, the method call algorithm can't really reliably choose mutable vs immutable deref. The challenge is that the proper choice will sometimes not be apparent until quite late in the process. For example, imagine the expression `p.foo()`: if `foo()` is defined with `&self`, we want an immutable deref, otherwise we want a mutable deref.

Note that in this RFC I do not *completely* address this issue. In particular, in an expression like `(*p).foo()`, where the dereference is explicit and not automatically inserted, the sense of the dereference is not inferred. For the time being, the sense can be manually specified by making the receiver type fully explicit: `(&mut *p).foo()` vs `(&*p).foo()`. I expect in a follow-up RFC to possibly address this problem, as well as the question of how to handle copies and moves of the referent (use #3 in my list above).

Lack of backtracking

Addressed by: New method resolution algorithm.

Issue #XYZ. When multiple traits define methods with the same name, it is ambiguous which trait is being used:

```
trait Foo { fn method(&self); }
trait Bar { fn method(&self); }
```

In general, so long as a given type only implements `Foo` *or* `Bar`, these ambiguities don't present a problem (and ultimately Universal Function Call Syntax or UFCS will present an explicit resolution). However, this is not guaranteed. Sometimes we see "blanket" impls like the following:

```
impl<A:Base> Foo for A { }
```

This impl basically says "any type `T` that implements `Base` automatically implements `Foo`". Now, we *expect* an ambiguity error if we have a type `T` that implements both `Base` and `Bar`. But in fact, we'll get an ambiguity error *even if* a type *only* implements `Bar`. The reason for this is that the current method resolution doesn't "recurse" and check additional dependencies when deciding if an `impl` is applicable. So it will decide, in this case, that the type `T` could implement `Foo` and then record for later that `T` must implement `Base`. This will lead to weird errors.

Overly conservative coherence

Addressed by: Expanded coherence rules.

The job of coherence is to ensure that, for any given set of type parameters, a given trait is implemented *at most once* (it may of course not be implemented at all). Currently, however, coherence is more conservative that it needs to be. This is partly because it doesn't take into account the very property that it itself is enforcing.

The problems arise due to the "blanket impls" I discussed in the previous section. Consider the following two traits and a blanket impl:

```
trait Base { }
trait Derived { }
impl<A:Base> Derived for A { }
```

Here we have two traits `Base` and `Derived`, and a blanket impl which implements the `Derived` trait for any type `A` that also implements `Base`.

This implies that if you implement `Base` for a type `S`, then `S` automatically implements `Derived`:

```
struct S;
impl Base for S { } // Implement Base => Implements Derived
```

On a related note, it'd be an error to implement *both* `Base` *and* `Derived` for the same type `T`:

```
// Illegal
struct T;
impl Base for T { }
impl Derived for T { }
```

This is illegal because now there are *two* implements of `Derived` for `T`. There is the direct one, but also an indirect one. We do not assign either higher precedence, we just report it as an error.

So far, all is in agreement with the current rules. However, problems arise if we imagine a type `U` that *only* implements `Derived`:

```
struct U;
impl Derived for U { } // Should be OK, currently not.
```

In this scenario, there is only one implementation of `Derived`. But the current coherence rules still report it as an error.

Here is a concrete example where a rule like this would be useful. We currently have the `Copy` trait (aka `Pod`), which states that a type can be memcopied. We also have the `Clone` trait, which is a more heavyweight version for types where copying requires allocation. It'd be nice if all types that could be copied could also be cloned – it'd also be nice if we knew for sure that copying a value had the same semantics as cloning it, in that case. We can guarantee both using a blanket impl like the following:

```
impl<T:Copy> Clone for T {
    fn clone(&self) -> T {
        *self
    }
}
```

Unfortunately, writing such an impl today would imply that no other types could implement `Clone`. Obviously a non-starter.

There is one not especially interesting ramification of this. Permitting this rule means that adding impls to a type could cause coherence errors. For example, if I had a type which implements `Copy`, and I add an explicit implementation of `Clone`, I'd get an error due to the blanket impl. This could be seen as undesirable (perhaps we'd like to preserve that property that one can *always* add impls without causing errors).

But of course we already don't have the property that one can always add impls, since method calls could become ambiguous. And if we were to add "negative bounds", which might be nice, we'd lose that property. And the popularity and usefulness of blanket impls cannot be denied. Therefore, I think this property ("always being able to add impls") is not especially useful or important.

Hokey implementation

Addressed by: Gradual vtable resolution algorithm

In an effort to improve inference, the current implementation has a rather ad-hoc two-pass scheme. When performing a method call, it will immediately attempt "early" trait resolution and – if that fails – defer checking until later. This helps with some particular scenarios, such as a trait like:

```
trait Map<E> {
    fn map(&self, op: |&E| -> E) -> Self;
}
```

Given some higher-order function like:

```
fn some_mapping<E,V:Map<E>>(v: &V, op: |&E| -> E) { ... }
```

If we were then to see a call like:

```
some_mapping(vec, |elem| ...)
```

the early resolution would be helpful in connecting the type of `elem` with the type of `vec`. The reason to use two phases is that often we don't need to resolve each trait bound to a specific impl, and if we wait till the end then we will have more type information available.

In my proposed solution, we eliminate the phase distinction. Instead, we simply track *pending constraints*. We are free to attempt to resolve pending constraints whenever desired. In particular, whenever we find we need more type information to proceed with some type-overloaded operation, rather than reporting an error we can try and resolve pending constraints. If that helps give more information, we can carry on. Once we reach the end of the function, we must then resolve all pending constraints that have not yet been resolved for some other reason.

Note that there is some interaction with the distinction between input and output type parameters discussed in the previous example. Specifically, we must never *infer* the value of the `Self` type parameter based on the impls in scope. This is because it would cause *crate concatenation* to potentially lead to compilation errors in the form of inference failure.

Properties

There are important properties I would like to guarantee:

- **Coherence** *or* **No Overlapping Instances:** Given a trait and values for all of its type parameters, there should always be at most one applicable impl. This should remain true even when unknown, additional crates are loaded.
- **Crate concatenation:** It should always be possible to take two creates and combine them without causing compilation errors. This property

Here are some properties I *do not intend* to guarantee:

- **Crate divisibility:** It is not always possible to divide a crate into two crates. Specifically, this may incur coherence violations due to the orphan rules.
- **Decidability:** Haskell has various sets of rules aimed at ensuring that the compiler can decide whether a given trait is implemented for a given type. All of these rules wind up preventing useful implementations and thus can be turned off with the `undecidable-instances` flag. I don't think decidability is especially important. The compiler can simply keep a recursion counter and report an error if that level of recursion is exceeded. This counter can be adjusted by the user on a crate-by-crate basis if some bizarre impl pattern happens to require a deeper depth to be resolved.

Detailed design

In general, I won't give a complete algorithmic specification. Instead, I refer readers to the prototype implementation. I would like to write out a declarative and non-algorithmic specification for the rules too, but that is work in progress and beyond the scope of this RFC. Instead, I'll try to explain in "plain English".

Method self-type syntax

Currently methods must be declared using the explicit-self shorthands:

```
fn foo(self, ...)
fn foo(&self, ...)
fn foo(&mut self, ...)
fn foo(~self, ...)
```

Under this proposal we would keep these shorthands but also permit any function in a trait to be used as a method, so long as the type of the first parameter is either `Self` or something derefable `Self`:

```
fn foo(self: Gc<Self>, ...)
fn foo(self: Rc<Self>, ...)
fn foo(self: Self, ...)      // equivalent to `fn foo(self, ...)
fn foo(self: &Self, ...)     // equivalent to `fn foo(&self, ...)
```

It would not be required that the first parameter be named `self`, though it seems like it would be useful to permit it. It's also possible we can simply make `self` not be a keyword (that would be my personal preference, if we can achieve it).

Coherence

The coherence rules fall into two categories: the *orphan* restriction and the *overlapping implementations* restriction.

Orphan check: Every implementation must meet one of the following conditions:

1. The trait being implemented (if any) must be defined in the current crate.
2. The `Self` type parameter must meet the following grammar, where `C` is a struct or enum defined within the current crate:

```
T = C
  | [T]
  | [T, ..n]
  | &T
  | &mut T
  | ~T
  | (..., T, ...)
  | X<..., T, ...> where X is not bivariant with respect to T
```

Overlapping instances: No two implementations of the same trait can be defined for the same type (note that it is only the `Self` type that matters). For this purpose of this check, we will also recursively check bounds. This check is ultimately defined in terms of the *RESOLVE* algorithm discussed in the implementation section below: it must be able to conclude that the requirements of one impl are incompatible with the other.

Here is a simple example that is OK:

```
trait Show { ... }
impl Show for int { ... }
impl Show for uint { ... }
```

The first impl implements `Show for int` and the case implements `Show for uint`. This is ok because the type `int` cannot be unified with `uint`.

The following example is *NOT OK*:

```
trait Iterator<E> { ... }
impl Iterator<char> for ~str  { ... }
impl Iterator<u8> for ~str { ... }
```

Even though `E` is bound to two distinct types, `E` is an output type parameter, and hence we get a coherence violation because the input type parameters are the same in each case.

Here is a more complex example that is also OK:

```
trait Clone { ... }
impl<A:Copy> Clone for A { ... }
impl<B:Clone> Clone for ~B { ... }
```

These two impls are compatible because the resolution algorithm is able to see that the type `~B` will never implement `Copy`, no matter what `B` is. (Note that our ability to do this check *relies* on the orphan checks: without those, we'd never know if some other crate might add an implementation of `Copy` for `~B`.)

Since trait resolution is not fully decidable, it is possible to concoct scenarios in which coherence can neither confirm nor deny the possibility that two impls are overlapping. One way for this to happen is when there are two traits which the user knows are mutually exclusive; mutual exclusion is not currently expressible in the type system [7] however, and hence the coherence check will report errors. For example:

```
trait Even { } // Naturally can't be Even and Odd at once!
trait Odd { }
impl<T:Even> Foo for T { }
impl<T:Odd> Foo for T { }
```

Another possible scenario is infinite recursion between impls. For example, in the following scenario, the coherence checked would be unable to decide if the following impls overlap:

```
impl<A:Foo> Bar for A { ... }
impl<A:Bar> Foo for A { ... }
```

In such cases, the recursion bound is exceeded and an error is conservatively reported. (Note that recursion is not always so easily detected.)

Method resolution

Let us assume the method call is `r.m(...)` and the type of the receiver `r` is `R`. We will resolve the call in two phases. The first phase checks for inherent methods [4] and the second phase for trait methods. Both phases work in a similar way, however. We will just describe how *trait method search* works and then express the *inherent method search* in terms of traits.

The core method search looks like this:

```
METHOD-SEARCH(R, m):
    let TRAITS = the set consisting of any in-scope trait T where:
        1. T has a method m and
        2. R implements T<...> for any values of Ty's type parameters

    if TRAITS is an empty set:
        if RECURSION DEPTH EXCEEDED:
            return UNDECIDABLE
        if R implements Deref<U> for some U:
            return METHOD-SEARCH(U, m)
        return NO-MATCH

    if TRAITS is the singleton set {T}:
        RECONCILE(R, T, m)

    return AMBIGUITY(TRAITS)
```

Basically, we will continuously auto-dereference the receiver type, searching for some type that implements a trait that offers the method `m`. This gives precedence to implementations that require fewer autodereferences. (There exists the possibility of a cycle in the `Deref` chain, so we will only autoderef so many times before reporting an error.)

Receiver reconciliation

Once we find a trait that is implemented for the (adjusted) receiver type `R` and which offers the method `m`, we must *reconcile* the receiver with the self type declared in `m`. Let me explain by example.

Consider a trait `Mob` (anyone who ever hacked on the MUD source code will surely remember Mobs!):

```
trait Mob {
    fn hit_points(&self) -> int;
    fn take_damage(&mut self, damage: int) -> int;
    fn move_to_room(self: GC<Self>, room: &Room);
}
```

Let's say we have a type `Monster`, and `Monster` implements `Mob`:

```
struct Monster { ... }
impl Mob for Monster { ... }
```

And now we see a call to `hit_points()` like so:

```
fn attack(victim: &mut Monster) {
    let hp = victim.hit_points();
    ...
}
```

Our method search algorithm above will proceed by searching for an implementation of `Mob` for the type `&mut Monster`. It won't find any. It will auto-deref `&mut Monster` to yield the type `Monster` and search again. Now we find a match. Thus far, then, we have a single autoderef `*victims`, yielding the type `Monster` – but the method `hit_points()` actually expects a reference (`&Monster`) to be given to it, not a by-value `Monster`.

This is where self-type reconciliation steps in. The reconciliation process works by *unwinding* the adjustments and adding auto-refs:

```
RECONCILE(R, T, m):
    let E = the expected self type of m in trait T;

    // Case 1.
    if R <: E:
      we're done.

    // Case 2.
    if &R <: E:
      add an autoref adjustment, we're done.

    // Case 3.
    if &mut R <: E:
      adjust R for mutable borrow (if not possible, error).
      add a mut autoref adjustment, we're done.

    // Case 4.
    unwind one adjustment to yield R' (if not possible, error).
    return RECONCILE(R', T, m)
```

In this case, the expected self type `E` would be `&Monster`. We would first check for case 1: is `Monster <: &Monster`? It is not. We would then proceed to case 2. Is `&Monster <: &Monster`? It is, and hence add an autoref. The final result then is that `victim.hit_points()` becomes transformed to the equivalent of (using UFCS notation) `Mob::hit_points(&*victim)`.

To understand case 3, let's look at a call to `take_damage`:

```
fn attack(victim: &mut Monster) {
    let hp = victim.hit_points(); // ...this is what we saw before
    let damage = hp / 10;         // 1/10 of current HP in damage
    victim.take_damage(damage);
    ...
}
```

As before, we would auto-deref once to find the type `Monster`. This time, though, the expected self type is `&mut Monster`. This means that both cases 1 and 2 fail and we wind up at case 3, the test for which succeeds. Now we get to this statement: "adjust `R` for mutable borrow".

At issue here is the overloading of the deref operator that was discussed earlier. In this case, the end result we want is `Mob::hit_points(&mut *victim)`, which means that `*` is being used for a *mutable borrow*, which is indicated by the `DerefMut` trait. However, while doing the autoderef loop, we always searched for impls of the `Deref` trait, since we did not yet know which trait we wanted. [2] We need to patch this up. So this loop will check whether the type `&mut Monster` implements `DerefMut`, in addition to just `Deref` (it does).

This check for case 3 could fail if, e.g., `victim` had a type like `Gc<Monster>` or `Rc<Monster>`. You'd get a nice error message like "the type `Rc` does not support mutable borrows, and the method `take_damage()` requires a mutable receiver".

We still have not seen an example of cases 1 or 4. Let's use a slightly modified example:

```
fn flee_if_possible(victim: Gc<Monster>, room: &mut Room) {
  match room.find_random_exit() {
    None => { }
    Some(exit) => {
      victim.move_to_room(exit);
    }
  }
}
```

As before, we'll start out with a type of `Monster`, but this type the method `move_to_room()` has a receiver type of `Gc<Monster>`. This doesn't match cases 1, 2, or 3, so we proceed to case 4 and *unwind* by one adjustment. Since the most recent adjustment was to deref from `Gc<Monster>` to `Monster`, we are left with a type of `Gc<Monster>`. We now search again. This time, we match case 1. So the final result is `Mob::move_to_room(victim, room)`. This last case is sort of interesting because we had to use the autoderef to *find* the method, but once resolution is complete we do not wind up dereferencing `victim` at all.

Finally, let's see an error involving case 4. Imagine we modified the type of `victim` in our previous example to be `&Monster` and not `Gc<Monster>`:

```
fn flee_if_possible(victim: &Monster, room: &mut Room) {
  match room.find_random_exit() {
    None => { }
    Some(exit) => {
      victim.move_to_room(exit);
    }
  }
}
```

In this case, we would again unwind an adjustment, going from `Monster` to `&Monster`, but at that point we'd be stuck. There are no more adjustments to unwind and we never found a type `Gc<Monster>`. Therefore, we report an error like "the method `move_to_room()` expects a `Gc<Monster>` but was invoked with an `&Monster`".

Inherent methods

Inherent methods can be "desugared" into traits by assuming a trait per struct or enum. Each impl like `impl Foo` is effectively an implementation of that trait, and all those traits are assumed to be imported and in scope.

Differences from today

Today's algorithm isn't really formally defined, but it works very differently from this one. For one thing, it is based purely on subtyping checks, and does not rely on the generic trait matching. This is a crucial limitation that prevents cases like those described in lack of backtracking from working. It also results in a lot of code duplication and a general mess.

Interaction with vtables and type inference

One of the goals of this proposal is to remove the hokey distinction between early and late resolution. The way that this will work now is that, as we execute, we'll accumulate a list of *pending trait obligations*. Each obligation is the combination of a trait and set of types. It is called an obligation because, for the method to be correctly typed, we must eventually find an implementation of that trait for those types. Due to type inference, though, it may not be possible to do this right away, since some of the types may not yet be fully known.

The semantics of trait resolution mean that, at any point in time, the type checker is free to stop what it's doing and *try* to resolve these pending obligations, *so long as none of the input type parameters are unresolved* (see below). If it is able to definitely match an impl, this may in turn affect some type variables which are *output type parameters*. The basic idea then is to always defer doing resolution until we either (a) encounter a point where we need more type information to proceed or (b) have finished checking the function. At those times, we can go ahead and try to do resolution. If, after type checking the function in its entirety, there are *still* obligations that cannot be definitely resolved, that's an error.

Ensuring crate concatenation

To ensure *crate concentanability*, we must only consider the `Self` type parameter when deciding when a trait has been implemented (more generally, we must know the precise set of *input* type parameters; I will cover an expanded set of rules for this in a subsequent RFC).

To see why this matters, imagine a scenario like this one:

```
trait Produce<R> {
    fn produce(&self: Self) -> R;
}
```

Now imagine I have two crates, C and D. Crate C defines two types, `Vector` and `Real`, and specifies a way to combine them:

```
struct Vector;
impl Produce<int> for Vector { ... }
```

Now imagine crate C has some code like:

```
fn foo() {
    let mut v = None;
    loop {
        if v.is_some() {
            let x = v.get().produce(); // (*)
            ...
        } else {
            v = Some(Vector);
        }
    }
}
```

At the point `(*)` of the call to `produce()` we do not yet know the type of the receiver. But the inferencer might conclude that, since it can only see one `impl` of `Produce` for `Vector`, `v` must have type `Vector` and hence `x` must have the type `int`.

However, then we might find another crate D that adds a new impl:

```
struct Other;
struct Real;
impl Combine<Real> for Other { ... }
```

This definition passes the orphan check because *at least one* of the types (`Real`, in this case) in the impl is local to the current crate. But what does this mean for our previous inference result? In general, it looks risky to decide types based on the impls we can see, since there could always be more impls we can't actually see.

It seems clear that this aggressive inference breaks the crate concatenation property. If we combined crates C and D into one crate, then inference would fail where it worked before.

If `x` were never used in any way that forces it to be an `int`, then it's even plausible that the type `Real` would have been valid in some sense. So the inferencer is influencing program execution to some extent.

Implementation details

The "resolve" algorithm

The basis for the coherence check, method lookup, and vtable lookup algorithms is the same function, called *RESOLVE*. The basic idea is that it takes a set of obligations and tries to resolve them. The result is four sets:

- *CONFIRMED*: Obligations for which we were able to definitely select a specific impl.
- *NO-IMPL*: Obligations which we know can NEVER be satisfied, because there is no specific impl. The only reason that we can ever say this for certain is due to the orphan check.
- *DEFERRED*: Obligations that we could not definitely link to an impl, perhaps because of insufficient type information.
- *UNDECIDABLE*: Obligations that were not decidable due to excessive recursion.

In general, if we ever encounter a NO-IMPL or UNDECIDABLE, it's probably an error. DEFERRED obligations are ok until we reach the end of the function. For details, please refer to the prototype.

Alternatives and downsides

Autoderef and ambiguity

The addition of a `Deref` trait makes autoderef complicated, because we may encounter situations where the smart pointer *and* its reference both implement a trait, and we cannot know what the user wanted.

The current rule just decides in favor of the smart pointer; this is somewhat unfortunate because it is likely to not be what the user wanted. It also means that adding methods to smart pointer types is a potentially breaking change. This is particularly problematic because we may want the smart pointer to implement a trait that *requires* the method in question!

An interesting thought would be to change this rule and say that we always *autoderef first* and only resolve the method against the innermost reference. Note that UFCS provides an explicit "opt-out" if this is not what was desired. This should also have the (beneficial, in my mind) effect of quelling the over-eager use of `Deref` for types that are not smart pointers.

This idea appeals to me but I think belongs in a separate RFC. It needs to be evaluated.

Footnotes

Note 1: when combining with DST, the `in` keyword goes first, and then any other qualifiers. For example, `in unsized RHS` or `in type RHS` etc. (The precise qualifier in use will depend on the DST proposal.)

Note 2: Note that the `DerefMut<T>` trait extends `Deref<T>`, so if a type supports mutable derefs, it must also support immutable derefs.

Note 3: The restriction that inputs must precede outputs is not strictly necessary. I added it to keep options open concerning associated types and so forth. See the Alternatives section, specifically the section on associated types.

Note 4: The prioritization of inherent methods could be reconsidered after DST has been implemented. It is currently needed to make impls like `impl Trait for ~Trait` work.

Note 5: The set of in-scope traits is currently defined as those that are imported by name. PR #37 proposes possible changes to this rule.

Note 6: In the section on autoderef and ambiguity, I discuss alternate rules that might allow us to lift the requirement that the receiver be named `self`.

Note 7: I am considering introducing mechanisms in a subsequent RFC that could be used to express mutual exclusion of traits.