rust-lang / **rust**

Tracking issue for RFC 1861: Extern types #43467

New issue

Open **aturon** opened this issue on 25 Jul 2017 · 85 comments

aturon commented on 25 Jul 2017 • edited by kennytm — Member

This is a tracking issue for RFC 1861 "Extern types".

Steps:

- [x] Implement the RFC (#44295)
- [ ] Adjust documentation (see instructions on forge)
- [ ] Stabilization PR (see instructions on forge)

Unresolved questions:

- Should we allow generic lifetime and type parameters on extern types? If so, how do they effect the type in terms of variance?
- In std's source, it is mentioned that LLVM expects `i8*` for C's `void*`. We'd need to continue to hack this for the two `c_void`s in std and libc. But perhaps this should be done across-the-board for all extern types? Somebody should check what Clang does.

aturon added **B-RFC-approved** **T-lang** labels on 25 Jul 2017

aturon changed the title from **Tracking issue for RFC 1861: Extern tyupes** to **Tracking issue for RFC 1861: Extern types** on 25 Jul 2017

aturon referenced this issue in **rust-lang/rfcs** on 25 Jul 2017

extern types #1861 — Merged

jethrogb commented on 25 Jul 2017 • edited — Contributor

This is not explicitly mentioned in the RFC, but I'm assuming different instances of `extern type` are actually different types? Meaning this would be illegal:

```
extern {
    type A;
    type B;
}

fn convert_ref(r: &A) -> &B { r }
```

canndrew commented on 25 Jul 2017 — Contributor

@jethrogb That's certainly the intention, yes.

glaebhoerl commented on 25 Jul 2017 • edited — Contributor

Assignees

No one assigned

Labels

B-RFC-approved
C-tracking-issue
T-lang

Projects

None yet

Milestone

No milestone

Notifications

22 participants

and others

Relatedly, is deciding whether we want to call it `extern type` or `extern struct` something that can still be done as part of the stabilization process, or is the `extern type` syntax effectively final as of having accepted the RFC?

EDIT: rust-lang/rfcs#2071 is also relevant here w.r.t. the connotations of `type` "aliases". In stable Rust a `type` declaration is "effect-free" and just a transparent alias for some existing type. Both `extern type` and `type Foo = impl Bar` would change this by making it implicitly generate a new module-scoped existential type or type constructor (nominal type) for it to refer to.

Ericson2314 commented on 25 Jul 2017 Contributor

Can we get a bullet for the panic vs DynSized debate?

👍 5

fitzgen referenced this issue in **rust-lang-nursery/rust-bindgen** on 25 Jul 2017

Tracking rustc bugs/features/RFCs that affect bindgen #849 Open

1 of 10 tasks complete

jimblandy referenced this issue on 26 Jul 2017

Poor error message for type definitions in `extern` blocks #43495 Open

Mark-Simulacrum added the **C-tracking-issue** label on 27 Jul 2017

tarcieri referenced this issue in **cesarb/clear_on_drop** on 5 Aug 2017

no_std support (closes #11) #12 Merged

plietar commented on 10 Aug 2017 Contributor

I've started working on this, and I have a working simple initial version (no generics, no DynSized).

I've however noticed a slight usability issue. In FFI code, it's frequent for raw pointers to be initialized to null using `std::ptr::null/null_mut`. However, the function only accepts sized type arguments, since it would not be able to pick a metadata for the fat pointer.

Despite being unsized, extern types are used through thin pointers, so it should be possible to use `std::ptr::null`.

It is still possible to cast an integer to an extern type pointer, but this is not as nice as just using the function designed for this. Also this can never be done in a generic context.

```
extern {
    type foo;
}
fn null_foo() -> *const foo {
    0usize as *const foo
}
```

Really we'd want is a new trait to distinguish types which use thin pointers. It would be implemented automatically for all sized types and extern types. Then the cast above would succeed whenever the type is bounded by this trait. Eg, the function `std::ptr::null` becomes :

```
fn null<T: ?Sized + Thin>() -> *const T {
    0usize as *const T
}
```

However there's a risk of more and more such traits creeping up, such as `DynSized`, making it confusing for users. There's also some overlap with the various custom RFCs proposals which allow arbitrary metadata. For instance, instead of `Thin`, `Referent<Meta=()>` could be used

SimonSapin commented on 10 Aug 2017 Contributor

I think we can add extern types now and live with `str::ptr::null` not supporting them for a while until we figure out what to do about `Thin` / `DynSized` / `Referent<Meta=…>` etc.

plietar commented on 10 Aug 2017 — Contributor

@SimonSapin yeah, it's definitely a minor concern for now.

I do think this problem of not having a trait bound to express "this type may be unsized be must have a thin pointer" might crop up in other places though.

SimonSapin commented on 11 Aug 2017 — Contributor

Oh yeah, I agree we should solve that eventually too. I'm only saying we might not need to solve all of it before we ship any of it.

plietar referenced this issue on 3 Sep 2017
Implement RFC 1861: Extern types #44295 — Merged

plietar commented on 3 Sep 2017 — Contributor

I've pushed an initial implementation in #44295

🎉 3 ❤️ 1

bors added a commit that referenced this issue on 4 Sep 2017
`Auto merge of #44295 - plietar:extern-types, r=arielb1` … `9dbee87`

crumblingstatue referenced this issue in **jeremyletang/rust-sfml** on 7 Sep 2017
[1.0] Commit to supporting a specific Rust version #148 — Open

bors added a commit that referenced this issue on 10 Sep 2017
`Auto merge of #44295 - plietar:extern-types, r=arielb1` … `abf1b10`

bors added a commit that referenced this issue on 13 Sep 2017
`Auto merge of #44295 - plietar:extern-types, r=arielb1` … `db66bed`

bors added a commit that referenced this issue on 1 Oct 2017
`Auto merge of #44295 - plietar:extern-types, r=arielb1` … `667de1d`

bors added a commit that referenced this issue on 28 Oct 2017
`Auto merge of #44295 - plietar:extern-types, r=arielb1` … `dce604a`

dtolnay referenced this issue in **dtolnay/syn** on 11 Nov 2017
Update AST to hold extern types #230 — Closed

kennytm referenced this issue on 15 Nov 2017
Support `extern type` in rustdoc. #46000 — Merged

kennytm commented on 16 Nov 2017 • edited — Member

@plietar In #44295 you wrote

> Auto traits are not implemented by default, since the contents of extern types is unknown. This means extern types are `!Sync`, `!Send` and `!Freeze`. This seems like the correct behaviour to me. Manual `unsafe impl Sync for Foo` is still possible.

While it is possible for Sync, Send, UnwindSafe and RefUnwindSafe, doing `impl Freeze for Foo` is *not* possible as it is a private trait in libcore. This means it is impossible to convince the compiler that an `extern type` is cell-free.

Should `Freeze` be made public (even if `#[doc(hidden)]`)? cc **@eddyb** #41349.

Or is it possible to declare an extern type is safe-by-default, which opt-out instead of opt-in?

```
extern {
    #[unsafe_impl_all_auto_traits_by_default]
    type Foo;
}
impl !Send for Foo {}
```

eddyb commented on 16 Nov 2017 • edited — Member

@kennytm What's the usecase? The semantics of `extern type` are more or less that of a hack being used before the RFC, which is `struct Opaque(UnsafeCell<()>);` , so the lack of `Freeze` fits.
That prevents rustc from telling LLVM anything *different* from what C signatures in clang result in.

kennytm commented on 16 Nov 2017 — Member

@eddyb Use case: Trying to see if it's possible to make CStr a thin DST.

I don't see anything related to a cell in #44295? It is reported to LLVM as an `i8` similar to `str` . And the places where `librustc_trans` involves the Freeze trait reads the real type, not the LLVM type, so LLVM treating all extern type as `i8` should be irrelevant?

eddyb commented on 16 Nov 2017 — Member

@kennytm So with `extern type CStr;` , writes through `&CStr` would be legal, and you don't want that?
The `Freeze` trait is private because it's used to detect `UnsafeCell` and not meant to be overriden.

kennytm commented on 16 Nov 2017 — Member

The original intent was to match the `extern type CStr` with the existing behavior of `struct CStr([c_char])` which is Freeze. Eddyb and I discussed on IRC, which assures that (1) Freeze is mainly used to disable certain optimizations only and (2) as Freeze is a private trait, no one other than the compiler will rely on it. So the missing Freeze trait will be irrelevant for `extern type CStr` .

parkovski commented on 18 Nov 2017

Regarding the thin pointer issue, I imagine that const generics will eventually enable constant comparisons in `where` clauses - if `size_of` is made const also, that would let you write a bound of `where size_of::<Ptr>() == size_of::<usize>()` which IMO matches the intent pretty perfectly.

👎 1

SimonSapin commented on 18 Nov 2017 — Contributor

This is the first I hear of allowing const expressions in where clauses. While it could be very useful, it seems far from given that this will be accepted into the language.

bors added a commit that referenced this issue on 18 Nov 2017

`Auto merge of #46000 - kennytm:fix-45640-extern-type-ice-in-rustdoc, …` ✓ `130eaae`

…

kennytm commented on 18 Nov 2017 • edited — Member

@SimonSapin Const expression in `where` clause will eventually be needed for const generics beyond RFC 2000 (spelled as `with` in rust-lang/rfcs#1932), but I do think this extension is out-of-scope for extern type or even custom DST in general.

parkovski commented on 18 Nov 2017

I didn't mean to assume that that will be supported, I meant that if it did become possible in a reasonable timeframe, which apparently is not likely, I think it'd be nice to have more regular syntax to express some ideas rather than more marker traits with special meaning given by the compiler. If that's not going to work, then great, it's one less thing to consider.

SSheldon referenced this issue in **SSheldon/rust-objc** on 6 Dec 2017

mem::swap doesn't work with Objects #6 Open

nox commented on 12 Dec 2017 Contributor

One of my use cases for extern types is to represent opaque things from the macOS frameworks with them.

For that purpose, I actually want to be able to wrap opaque things in some generic types to encode certain invariants related to refcounting.

For example, I want a `CFRef<T>` type that derefs to `CFShared<T>` that itself derefs to `T`.

```
pub struct CFRef<T>(*const CFShared<T>);
pub struct CFShared<T>(T);
```

This is apparently not possible if `T` is an extern type.

```
pub extern type CFString;
fn do_stuff_with_shared_string(str: &CFShared<CFString>) { ... }
```

Would it be complicated to support such a thing?

kennytm commented on 12 Dec 2017 Member

@nox `struct CFShared<T: ?Sized>(T);` ?

Note that this may not compile after we have implemented the `DynSized` trait since an `extern type` is not `DynSized`, and we can't place a `!DynSized` field inside a struct (may need explicit `#[repr(transparent)]` to allow it.

gnzlbg commented on 3 Feb • edited Contributor

Can somebody provide an update on the status of this and maybe summarize the remaining open issues? I'd like to know if it is possible already to implement `c_void` in `libc` / `core` using `extern type`, for example.

main-- referenced this issue in **main--/rustgres** on 3 Feb

extern type varlena #6 Open

jethrogb referenced this issue in **rust-lang-nursery/portability-wg** on 18 Mar

libc dependencies #13 Open

jethrogb commented on 18 Mar • edited Contributor

@gnzlbg as mentioned in the initial post in this issue:

> In std's source, it is mentioned that LLVM expects `i8*` for C's `void*`.
> We'd need to continue to hack this for the two `c_void`s in std and libc.
> But perhaps this should be done across-the-board for all extern types?
> Somebody should check what Clang does.

There is no solution for this yet, I think.

Ericson2314 commented on 18 Mar • edited Contributor

@gnzlbg Whether we want `DynSized` needs to be resolved as well. The description of the initial implementation does a great job of laying out the footguns that exist without it. Thanks **@plietar**!

jethrogb commented on 18 Mar • edited Contributor

C Opaque struct:

```
typedef struct c_void c_void;

c_void* malloc(unsigned long size);

void call_malloc() {
        malloc(1);
}
```

clang version 5.0.1:

```
%struct.c_void = type opaque

define void @call_malloc() #0 {
  %1 = call %struct.c_void* @malloc(i64 1)
  ret void
}

declare %struct.c_void* @malloc(i64) #1
```

C void:

```
void* malloc(unsigned long size);

void call_malloc() {
        malloc(1);
}
```

clang version 5.0.1:

```
define void @call_malloc() #0 {
  %1 = call i8* @malloc(i64 1)
  ret void
}

declare i8* @malloc(i64) #1
```

Rust extern type:

```
#![feature(extern_types)]
#![crate_type="lib"]

extern "C" {
    type c_void;

    fn malloc(n: usize) -> *mut c_void;
}

#[no_mangle]
pub fn call_malloc() {
    unsafe { malloc(1); }
}
```

Rust nightly:

```
%"::c_void" = type {}

define void @call_malloc() unnamed_addr #0 !dbg !4 {
start:
```

```
  %0 = call %"::c_void"* @malloc(i64 1), !dbg !8
  br label %bb1, !dbg !8

bb1:                                              ; preds = %start
  ret void, !dbg !10
}

declare %"::c_void"* @malloc(i64) unnamed_addr #1
```

Ericson2314 commented on 18 Mar — Contributor

Great find! This `%struct.c_void = type opaque` sure looks worth imitating. If that's slower than `i8*` IMO that's a clang/LLVM bug to report.

jethrogb commented on 18 Mar — Contributor

Would this syntax be acceptable?

```
extern {
    #[repr(i8)]
    type c_void;
}
```

👍 2

Ericson2314 commented on 18 Mar — Contributor

@jethrogb looks good to me, but I'd be tempted to keep it unstable as it only exists to hack around LLVM.

jethrogb commented on 18 Mar — Contributor

> I'd be tempted to keep it unstable as it only exists to hack around LLVM.

That sounds good in principle, but I think there were plans to have the final public definition of `c_void` live in a crates.io crate.

Ericson2314 commented on 18 Mar • edited — Contributor

@jethrogb hehe just quoted those plans. I do see the tension, bummer.

whitequark commented on 18 Mar — Contributor

> Great find! This %struct.c_void = type opaque sure looks worth imitating. If that's slower than i8* IMO that's a clang/LLVM bug to report.

It is definitely slower than `i8*`. For example, malloc declared as returning an opaque struct won't be recognized as malloc by most (all?) optimization passes. It's a known issue and according to some LLVM devs the right way to fix it is to get rid of pointee types altogether and have just one pointer type, but that doesn't seem to be happening any time soon so you'll have to emit `i8*`.

👍 2

gnzlbg commented on 19 Mar — Contributor

@whitequark do you happen to have a link to the LLVM bug?

whitequark commented on 19 Mar — Contributor

@gnzlbg I'm not sure if there's one, that was from IRC discussions.

gnzlbg commented on 19 Mar • edited ▾ Contributor

I've tried to search for one without any luck so I've filled this one: https://bugs.llvm.org/show_bug.cgi?id=36795

What **@whitequark** pointed out looks correct, when using i8* LLVM can eliminate calls to malloc, while when using a type opaque c_void* it cannot.

🎉 1

SimonSapin commented on 28 Mar Contributor

In the RFC:

> As a DST, `size_of` and `align_of` do not work, but we must also be careful that `size_of_val` and `align_of_val` do not work either, as there is not necessarily a way at run-time to get the size of extern types either. For an initial implementation, those methods can just panic, but before this is stabilized there should be some trait bound or similar on them that prevents their use statically. The exact mechanism is more the domain of the custom DST RFC, RFC 1524, and so figuring that mechanism out will be delegated to it.

However RFC 1524 was closed. Its successor is probably rust-lang/rfcs#2255, but that's an issue rather than a PR for a new RFC.

Per #46108 (comment) the lang team recently decided *against* having a `DynSized` trait. But that leaves an unresolved question in this open RFC.

In rustc 1.26.0-nightly (`9c9424d` 2018-03-27), this compiles without warning and prints `0` :

```
#![feature(extern_types)]

extern { type void; }

fn main() {
    let x: *const void = main as *const _;
    println!("{}", std::mem::size_of_val(unsafe { &*x }));
}
```

The libs team discussed defining a public `void` extern type in the standard library and changing the return type of memory allocation APIs to `*mut void` instead of `*mut u8` . However in that case we'd need to decide what to do about `size_of_val` + extern types before allocations APIs are stabilized. (Keeping `void` unstable wouldn't help, if you can obtain a pointer to it you don't need to name the type to call `size_of_val` .)

CC **@rust-lang/lang**

kennytm commented on 28 Mar Member

rust-lang/rfcs#1524 (Custom DST) is orthogonal to rust-lang/rfcs#2255 (Whether we want more `?Trait`). The successor is https://internals.rust-lang.org/t/pre-erfc-lets-fix-dsts/6663.

In #46108 we decided against `?DynSized` , but I think a `DynSized` without a `?` (e.g. rust-lang/rfcs#2310 or rust-lang/rfcs#2255 (comment)) is still on the table.

BTW for consistency with common C extensions, if the size_of `void` cannot be undefined, it should be set to 1.

joshtriplett commented on 30 Mar Member

Conclusions from the lang meeting at the all-hands:

- `size_of_val` should panic if called on an `extern type`
- We should have a best-effort lint to statically detect if you call `size_of_val` on an `extern type` , either directly or ideally also through a generic.
- None of this impacts the ability to do custom DSTs.

Does anyone have a specific good reason this shouldn't panic, and should instead abort?

👍 2

gnzlbg commented on 30 Mar — Contributor

> ideally also through a generic.

Would this result in a monomorphization time error?

cramertj commented on 30 Mar — Member

@gnzlbg It sounded like folks were generally in favor of a monomorphization-time lint.

👍 1 😕 1

glaebhoerl commented on 30 Mar — Contributor

Would appreciate if someone could also elaborate on the reasoning behind these conclusions. :)

joshtriplett commented on 30 Mar — Member

Rough summary: `extern type` is a special-purpose feature that exists for FFI, so adding a pile of trait machinery to statically detect and reject calls to `size_of_val` on one didn't seem worth it. We had a *very* strong consensus against returning a sentinel value, which left us with "either panic or abort". There was some discussion about whether we had any motivation to abort, but we couldn't think of any specific cases where panic would lead to breakage. Finally, we still do like the idea of statically detecting these issues, but we can do that with a lint for at least the most common cases.

👍 1

nikomatsakis commented on 30 Mar — Contributor

@glaebhoerl Hey =) It's kind of hard to write up a detailed comment just now, but I want to say a few things. **First off, like any weighty decision, I would describe this as a "preliminary conclusion", subject as always to revision if persuasive counterarguments arise.** =)

As for reasoning, there are some minutes from discussion here but they're pretty brief. Here is my attempt at a summary of the key points as I remember them:

- The "desugaring" of `T: ?Sized` works is already fairly surprising to users as is.
  - Extending to a three-layer hierarchy makes it quite tongue twisting even for advanced users:
    - You say `T` to mean `T: Sized`
    - You say `T: ?Sized` to mean `T: DynSized`
    - You say `T: ?DynSized` to mean `T`
- We would like to extend to custom DSTs in the future; this is often cited as being connected to `DynSized`, but that doesn't seem entirely complete. We can still have a `DynSized` trait (or family of traits), but they don't have to be supplied by default:
  - If you write `T`, you get `Sized` so you're all set
  - If you write `T: ?Sized`, you get nothing, but have to add other bounds just like ordinary bounds
    - it does mean that `size_of_val` and `align_of_val` are always invokable for any type
      - but these are the most general case anyway (when you have the full value + its metadata); we're covering the hard case now.
- When we have custom DST, that implies that `size_of_val` and friends will run user code *anyway*. That code could panic.
  - Given that, we will have the possibility of `size_of_val` panicking.
- There was some *mild* concern that `size_of_val` might execute in unsafe code that is not panic safe, creating a footgun.
  - We *could* make it hard abort instead -- also if user code panics.
  - But we wanted more persuasive arguments, e.g. examples of code in the wild that would have a problem (brief inspection of code in the standard library didn't turn up such problems, but I didn't really look especially hard).

👍 2

glaebhoerl commented on 30 Mar

Contributor

Thanks!

(To be clear I'm skeptical about the value of `?DynSized` as well, at least on its own, when its *only* utility would be to prevent misuse of `size_of_val`.)

I was mainly curious about the reasoning around the choice of "panic" versus "return 0". I don't think of `0` as being a sentinel value in this case, if "sentinel value" is understood as "something the caller has to check for specifically and handle specially". I agree that panicking is preferable to this.

I think of `0` as a "safest possible default value" -- that is, if someone asks for the `size_of_value` of an `extern type`, gets `0`, and proceeds to read and write 0 bytes to and from memory, the effect will be that of a no-op, which is unlikely to actually cause any problems. The question is what (if any) scenarios are there where it *would*. (I might have asked this same question on the `extern type` RFC thread and someone might have even tried to answer it...)

kennytm commented on 30 Mar

Member

Note that making `align_of_val` panic also means that field access will potentially panic:

```
extern { type Opaque; }
struct TerribleOpaque {
    a: u8,
    b: Opaque,
}

let a: &TerribleOpaque = unsafe { ... };
let b: &Opaque = &a.b;   // <-- this line will panic.

struct GenericThingy<T: ?Sized> {
    c: u8,
    d: T,
}
let c: &GenericThingy<Opaque> = ...;
let d: &Opaque = &c.d;   // <-- this line will also panic.
```

joshtriplett commented on 30 Mar

Member

Returning a dummy value rather than asserting/panicking seems really unlike Rust, and not something we typically do. We don't do things like returning `-1`; we use `Option`, or we panic or assert.

👍 1

Ericson2314 commented on 30 Mar • edited

Contributor

@nikomatsakis could we keep this unstable until we have a custom DST experiment then? Or we could stabilize this with unstable DynSized which would just prevent generics over extern types in practice which is probably fine, while allowing it to be removed later based on DST experiment

> but these are the most general case anyway (when you have the full value + its metadata); we're covering the hard case now.

You mean custom DSTs in practice would all implement DynSized?

> Given that, we will have the possibility of size_of_val panicking.

Sure any code may panic, but using a panic to enforce a static invariant still leaves a bitter taste in month. If a library makes a "false instance" that is considered bad form. This only is different because of concerns about opt-in traits which may tip the scales in aggregate but doesn't address this problem.

I want a solution that doesn't feel born out of tragic trade-offs.

joshtriplett commented on 30 Mar

Member

@Ericson2314 The expectation from the discussion was that a lint ought to be able to catch the vast majority of such cases.

whitequark commented on 30 Mar Contributor

> I think of 0 as a "safest possible default value" -- that is, if someone asks for the size_of_value of an extern type, gets 0, and proceeds to read and write 0 bytes to and from memory, the effect will be that of a no-op, which is unlikely to actually cause any problems.

Let's say I'm trying to serialize a value (in a generic function, as it goes) somewhere and use `size_of_val` for that. Now, when I deserialize it, I have a problem.

glaebhoerl commented on 30 Mar Contributor

@joshtriplett I agree, and the point of my comment was to explain why I think this is unlike that.

@whitequark Thanks. To have a problem, you'd need the deserialization code to somehow derive a different value? How could/would that end up happening? (I guess if the deserialization happens in C? But then why is the Rust code using generics to hand-roll its own serialization instead of calling C?)

(I just want to be duly diligent and identify, as an existence proof (or 'smoking gun'), at least one plausible, concrete real-world scenario where this causes a major problem before we judge that it's 'obviously' a bad idea.)

whitequark commented on 30 Mar • edited Contributor

@glaebhoerl

```
fn serialize<T>(storage: &mut [u8], val: &T) {
  let size = mem::size_of_val(val);
  storage[..size].copy_from_slice(slice::from_raw(val as *const T as *const u8, size));
}

fn deserialize<T>(storage: &[u8], val: &mut T) {
  let size = mem::size_of_val(val);
  slice::from_raw_mut(val as *mut T as *mut u8, size).copy_from_slice(&storage[..size]);
}

extern {
  type Foo;
  fn alloc_foo() -> *mut Foo;
}

// somewhere:
let original_foo: &Foo = ...;
let new_foo: &mut Foo = unsafe { alloc_foo() as &mut Foo };
let buf: &mut [u8] = ...;
serialize(buf, original_foo);
deserialize(buf, new_foo);
// now new_foo contains uninitialized data.
```

❤️ 1

glaebhoerl commented on 30 Mar Contributor

Ah I see. In that case the part which 'does know' the size is `alloc_foo`, which sounds realistic enough. I'm convinced, thanks again!

glaebhoerl commented on 30 Mar Contributor

Unrelatedly, I want to re-raise the question of whether we want to deprecate `size_of_val` and replace it with something which returns an `Option`. That would take some of the edge off of `size_of_val` panicking which nobody *likes*.

joshtriplett commented on 30 Mar Member

> On March 30, 2018 6:07:02 PM GMT+02:00, "Gábor Lehel" ***@***.***> wrote:
> Unrelatedly, I want to re-raise the question of whether we want to
> deprecate `size_of_val` and replace it with something which returns an
> `Option`. That would take some of the edge off of `size_of_val`
> panicking which nobody *likes*.

That would make size_of_val significantly less useful, and in practice would lead to a lot of unwrapping (which just panics anyway).

It only fails when called on something it should never get called on, and we can detect that case with a lint.

👍 3

SimonSapin commented on 30 Mar — Contributor

> `extern type` is a special-purpose feature that exists for FFI

> could we keep this unstable until we have a custom DST experiment then

The libs team hopes to stabilize relatively soon (a subset of) allocator APIs after changing them to use `*mut void` to represent pointers to allocated memory, with `void` (name to be bikeshedded) an extern type. The type being `!Sized` is valuable to prevent the use of `<*mut _>::offset` without first casting to another pointer type, but the pointers must be thin.

So while FFI was a primary motivation it's not the only case when extern types might show up, and it would be nice to be able to stabilize them without waiting for a full design for custom DSTs.

RalfJung commented on 31 Mar • edited — Contributor

> I think of 0 as a "safest possible default value" -- that is, if someone asks for the size_of_value of an extern type, gets 0, and proceeds to read and write 0 bytes to and from memory, the effect will be that of a no-op, which is unlikely to actually cause any problems.

If you are ever asking for the size of an `extern type`, something has gone awfully wrong somewhere. This size is by definition not knoweable. `0` is most certainly not a safe choice if e.g. the offset is used to get an address that definitely lives "after" the extern data in memory; the code would instead overwrite that data which is rather not a safe choice.

~~**@whitequark** how would reporting the (incorrect!) size `0` be *helpful* with deserialization? If you attempt to deserialize a type of which you do not know the size, and that deserialization somehow needs the size, then you are kind of in a hard place and something went wrong somewhere. "Just go on and pretend nothing happened" is not how Rust solves these kinds of problems.~~

@kennytm How does Rust even compute the layout of a struct like

```
struct TerribleOpaque {
    a: u8,
    b: Opaque,
}
```

given that rustc does not know the alignment of `Opaque` either? I expect such type definitions to be illegal. And vice versa, if rustc somehow *does* come up with a layout and a choice for the offset of `b`, then it can just use that value when doing `&c.b` at run-time. Field access will never panic; it compiles (AFAIK) to a constant-offset operation because the offset of the field is computed at compile-time, never at run-time.

👍 1 ❤️ 1

kennytm commented on 31 Mar • edited — Member

@RalfJung

> Field access will never panic; it compiles (AFAIK) to a constant-offset operation because the offset of the field is computed at compile-time, never at run-time.

No the offset `&c.b` can be computed at run-time when the field is a DST. Check this:

```
let y16: &GenericThingy<dyn Debug> = &GenericThingy { c: 10u8, d: 20u16 };
```

```
let y32: &GenericThingy<dyn Debug> = &GenericThingy { c: 30u8, d: 40u32 };
assert_eq!(
    (&y16.d as *const _ as *const u8 as usize) - (y16 as *const _ as *const u8 as usize),
    2
);
assert_eq!(
    (&y32.d as *const _ as *const u8 as usize) - (y32 as *const _ as *const u8 as usize),
    4
);
```

Although `y16` and `y32` have the same type, the offset of `&self.d` is different.

Currently, the offset of this DST field `d: T` of a DST struct `GenericThingy<T>` is computed by the compact-size-of the sized prefix, rounded up to the alignment of the DST field type `T`. Therefore, to compute the offset of the field, we must require the type `T` to have an alignment derivable from its metadata only. In the Custom DST proposal this means `T: AlignFromMeta`.

So yes `TerribleOpaque` is illegal. However,

1. The definition `GenericThingy` is clearly legal,
2. **Since there is no `DynSized` or `AlignFromMeta`, there is nothing blocking us from instantiating `GenericThingy<Opaque>`**
3. Unless you introduce post-monomorphization error, `&c.d` should have the same compile-time behavior whether it is `GenericThingy<u8>`, `GenericThingy<[u8]>` or `GenericThingy<Opaque>`.

This means either `&c.d` must panic at runtime, or choose a fallback alignment such as 1 or `align_of::<usize>()`.

❤️ 4

nikomatsakis commented on 31 Mar • edited ▾ Contributor

@Ericson2314

> could we keep this unstable until we have a custom DST experiment then?

I'm not inclined to wait. As **@SimonSapin** said, the FFI need is real now. Also, I'd like to drill into what **specific** choices around Custom DST are being forced here. It seems that what we are deciding is actually relatively narrow:

Will we try to narrow the range of types on which you can invoke `size_of_val`?

We are leaning towards "no" on that particular question, but that does not necessarily imply that all custom DST types must implement `DynSized`. We might, for example, say that `size_of_val` and friends use specialization to check for what sort of trait the reference implements (if any) and panic if there is no such trait implemented -- presuming that always applicable impls work out like I think they will, that would be eminently doable (and **@aturon** had an exciting idea for building on that work, too, that helps here).

Even if we did say that everything must implement `DynSized`, then it seems like we are distinguishing a class of types (including at least `extern type`) for which said implementations unconditionally panic. We are saying that it is not worth distinguishing that classic soundly in the trait system, but we could use lints to capture that class when generics are not involved. (And go further with monomorphization-time lints, if desired.)

nikomatsakis commented on 31 Mar Contributor

@kennytm

> Note that making align_of_val panic also means that field access will potentially panic:

Yeah, good point! This would also be a consequence of custom DST. I think strengthens the case for "hard abort" and not panic -- it seems like predicting which field accesses could panic would be quite subtle, and a potential optimization hazard. (It may also argue for a monomorphization-time lint.)

If we did opt for "hard abort" instead of panic, I would say that the rule is:

> Custom DST code is not permitted to panic (much like panicking across an FFI boundary). We will dynamically capture such panics and convert them into a hard abort.

❤️ 1

nikomatsakis commented on 31 Mar • edited Contributor

UPDATE: Ignore this, it doesn't work because of back-compat; you can do coercions in a generic context, obviously.

Hmm I wonder if we can modify the definition of `CoerceUnsized` to prevent "unsizing" a final field into an extern type altogether? That would, I believe, avoid the concern about field access (and moves more towards the specialized-based interpretation of `size_of_val` I proposed here). ~~

That is, we might have a trait `DynSized`, which is not implemented for `extern type`, and we say that you cannot use coerce unsized unless the target type implements it. But it is not required to invoke `size_of_val`.

nikomatsakis commented on 31 Mar Contributor

@glaebhoerl

> Unrelatedly, I want to re-raise the question of whether we want to deprecate size_of_val and replace it with something which returns an Option. That would take some of the edge off of size_of_val panicking which nobody likes.

I see this as a separate question, but I am sympathetic to your desire. That said, I agree also with **@joshtriplett** that in many cases one will just unwrap the result -- at minimum, we ought to add some sort of function to readily *test* if a type has a defined size/alignment, so you *can* code defensively (even if we are not going to say you must).

RalfJung commented on 31 Mar Contributor

> This means either &c.d must panic at runtime, or choose a fallback alignment such as 1 or align_of::().

Thanks, I clearly had not thought this through enough.

However, returning any arbitrary (and hence wrong!) fixed alignment seems catastrophic for the case you described -- it would let us compute the wrong address, as C code that knows the actual extern type could end up with a different layout than we do. So, doesn't your example show that we *have to* panic or abort?

> I think strengthens the case for "hard abort" and not panic -- it seems like predicting which field accesses could panic would be quite subtle, and a potential optimization hazard. (It may also argue for a monomorphization-time lint.)

Agreed -- not just because of optimizations, but also because unsafe code has to be very aware where panics could be raised, for exception safety purposes.

👍 2

gnzlbg commented on 31 Mar • edited Contributor

Would these interact in any way with NVPTX `extern __shared__` array types ? There are two flavors of `__shared__` array types, static (non `extern`) and dynamic (`extern`).

```
fn kernel() {
   let mut a: #[shared] [f32; 16];
   // ^^ This array is shared by all threads in a thread-group
   // It's size is fixed at compile-time and it is the same for
   // all kernel invocations.

   let mut b: #[shared] [u8];
   // ^^ This array is shared by all threads in a thread-group.
   // It has a dynamic size that is constant during the invocation
   // of this kernel. Each kernel launch must set its size, but each time
   // this kernel is launched this array can have a different length. This
   // basically produces a pointer. The user is responsible for tracking
   // the size of these arrays, e.g., by passing it as an argument to the
   // kernels.
}
```

So it looks to me that this wouldn't interact with static `__shared__` arrays because `size_of_val` would just return `mem::size_of` . However, the size of `extern __shared__` arrays is not known, not at compile-time, and at least for nvptx not at run-time either: the user is in charge of passing the array size around as a kernel argument and it is a "common" idiom to pass a single integer from which multiple sizes are computed inside the kernel. So I assume `size_of_val` would need to result in an error for these.

arielb1 commented on 31 Mar • edited Contributor

Field access being potentially aborting feels very sad.

On the other hand, the only way this can happen is when the struct type is uninhabitable, in which case the field access was dubious anyway. So this basically raises the old question of when is calling `size_of_val` or doing raw pointer lvalue access is valid.

I would prefer that to be documented somewhere - obviously, we want to access fields of e.g. uninitialized structs, as in e.g. `RcFromSlice` .

canndrew commented on 2 Apr Contributor

The recent `DynSized` RFC proposed

- `T` to mean `T: Sized`
- `T: ?Sized` to mean `T` with no trait bounds
- `T: ?Sized + DynSized` to mean `T: DynSized`

Where `extern` types would be `!DynSized` , but then only adding `+ DynSized` to `size_of_val` in the new epoch, leaving it as a lint+panic for now. Since this is an extension of what's being proposed here and could be added later, is the idea still on the table?

SimonSapin commented on 2 Apr Contributor

Since we want to compile together crates that use different epochs/editions, opting into a new edition can only affect "superficial" crate-local aspects of the language like syntax, not public APIs.

Ericson2314 commented on 3 Apr • edited Contributor

@nikomatsakis

My basic concern is I feel a number of various issues are pushing us in the direction of more fundamental / opt-in traits, but the resistance to opt-in traits is such that we're throwing around ~~ad-hoc lints~~ ad-hoc solutions like lints instead. I get that `?` is annoying to teach, but I'll take principled weirdness over banal but endless machinations. The grapple scares me more than the fall down this slippery slope.

I'll admit `{size,align}_of_val` isn't that interesting on it's own. But to show my cards, I was excited about contributing in part to this RFC because I finally had some issue by which to force the topic of `DynSized` in particular, and more special traits in general. I guess you all didn't take the bait :). Now, I suppose I'll ask whether, if we had a full menagerie already, would we still bother making `{size,align}_of_val` defined for `!DynSized` types. Relatedly, if we end up adding `DynSized` later, would be deprecate the `{size,align}_of_val` we have today? I realize "no" for the first and "yes" for the second aren't ironclad reasons to make `DynSized` now, but I'm still curious about the answer.

> Even if we did say that everything must implement `DynSized`, then it seems like we are distinguishing

Mmm if all types must implement `DynSized`, then we're *not* distinguishing anything. What we are doing is providing a principled way of using an existing feature (the trait system) to allow users to right the requisite "hook". That alone is reason for a `DynSized` trait in my mind.

> ...a class of types (including at least `extern` type) for which said implementations unconditionally panic.

Surely you don't mean the salient attribute of `!DynSized` types is that querying the size panics? It's that they have no dynamically or statically known size. Panicking is just an enforcement mechanism with no intrinsic meaning.

canndrew commented on 3 Apr — Contributor

> opting into a new edition can only affect "superficial" crate-local aspects of the language like syntax, not public APIs.

We could deprecate and replace `size_of_val` then. Call it `dynsize_of_val`.

Ericson2314 referenced this issue on 4 Apr

Add DynSized trait (rebase of #44469) #46108 — Closed

This was referenced on 4 Apr

Tracking issue for the GlobalAlloc trait and related APIs #49668 — Closed

Add GlobalAlloc trait + tweaks for initial stabilization #49669 — Merged

joshtriplett referenced this issue on 5 Apr

`extern type` cannot support `size_of_val` and `align_of_val` #49708 — Open

nikomatsakis commented on 5 Apr — Contributor

@canndrew (I am responding to two comments at once)

> The recent `DynSized` RFC proposed
>
> - `T` to mean `T: Sized`
> - `T: ?Sized` to mean `T` with no trait bounds
> - `T: ?Sized + DynSized` to mean `T: DynSized`
>
> The recent DynSized RFC proposed ... We could deprecate and replace `size_of_val` then. Call it `dynsize_of_val`.

I believe that this future could still be on the table. This is what I was trying to say in this comment when I wrote:

> It seems that what we are deciding is actually relatively narrow:
>
> **Will we try to narrow the range of types on which you can invoke `size_of_val`?**

I feel very strongly that we do not want `T: ?Sized` to actually mean `T: DynSized`. However, I *could* imagine that we introduce `DynSized` as an "ordinary" trait and introduce `dynsize_of_val` (or whatever) that requires it -- and then specify that `size_of_val` is implemented by using specialization to invoke `dynsize_of_val` when possible and aborting/packing otherwise (I lean more and more towards abort, personally).

Alternatively, thinking more about lints -- it is certainly plausible to lint on calls to `size_of_val` *unless* `T: DynSized` (one could even imagine generalizing this). That is important because we also do have to figure out the field access question. We can't deprecate field accesses -- and they are legal today knowing only that `T: ?Sized` (i.e., we do not require `T: DynSized`). But we could lint aggressively there, thus encouraging `T: DynSized` to proliferate.

Worth thinking over. But also not blocking further progress on `extern type`, I think.

❤️ 1

nikomatsakis commented on 5 Apr

Contributor

@Ericson2314

> My basic concern is I feel a number of various issues are pushing us in the direction of more fundamental / opt-in traits, but the resistance to opt-in traits is such that were throwing around ad-hoc lints instead.

Can you be more explicit? It seems like this is one precise case where we are talking about lints, specifically because it is narrow and we don't see another way out of the backwards compatibility box, but in other cases where we had thought about adding "implicit" traits (notably, `?Move`), I don't believe lints are on the table. Instead, we've found a way to add the desired functionality in a "non-infectious" fashion (using `Pin`). Are there other cases I'm overlooking?

That said, I *do* think there is a constant tension, one that Rust always has to walk: how to get the maximum bang for our static analysis buck, and I feel no shame about keeping lints as part of the toolbox.

Ericson2314 commented on 6 Apr • edited

Contributor

@nikomatsakis

> Can you be more explicit?

Sorry, I meant "ad-hoc solutions like lints". edited the above accordingly.

I hadn't yet seen `Pin`. Glad there is a safe and total way out of that corner, but it too strikes me as a bit of a monkey patch; see the final comment rust-lang/rfcs#2349 (comment) which makes one wonder whether all collections will need a `Pin` variant leading to an ecosystem split!

I realize there's a steep drop off in priority along [generators, extern types, custom DSTs, out pointers and other linear types]. But the fact that implicit traits keep coming up gives me pause to let them go: I now see them all as one problem and thus our current trajectory as many unrelated piecemeal solutions. Also, the observation (not mine, maybe in rust-lang/rfcs#2255 ?) that *more* `?`-traits probably makes them *less* confusing I find compelling.

Also, `?`-traits work like Cargo features in that their the only general way to backwards-compatibly grow the language in a *negative* direction: reducing requirements rather than adding functionality, and I find that the more interesting direction for language evolution.

This all boils down to a difference in but opinion that's been around for years, and probably cannot be bridged. Your previous comment on positive `DynSized` gives me hope in this specific case. If we have far more `DynSize` than `?DynSize` annotations in the end, I wonder what is achieved, but at least we can meaningfully speak about sizing.

Ericson2314 referenced this issue in **rust-lang/rfcs** on 6 Apr

More implicit bounds (?Sized, ?DynSized, ?Move) #2255 Open

mikeyhew commented on 6 Apr

Contributor

OK, I just want to say that I am **very strongly** of the opinion that Rust should use built-in traits like `DynSized` to express the difference in capabilities between `extern` types and the dynamically-sized types that currently exist in Rust (i.e. trait objects and slices). All of the alternatives that I have seen – panicking, returning `Option<usize>` from `size_of_val`, post-monomorphisation lints – are less powerful, and the issues with `?`-traits that people keep bringing up *need* to be tested and not just speculated about. We need to at least *try* doing things the builtin-traits way and see what it's like, and see what the ergonomic impact is like, and see if we can reduce it, before settling for something inferior.

Maybe I'm overreacting, I just got the sense from reading some of the comments in this thread that something might be done in order to get `extern` types out the door, that might put us in a backward-compatibility trap later on. Now that I have more time to work on Rust, I'm planning on writing an eRFC to add DST-specific builtin traits like `DynSized` and `SizeFromMeta`, so we can start experimenting with them and Custom DST.

👍 3 ❤️ 2

aturon commented on 6 Apr

Member

@mikeyhew These alternatives are definitely less powerful, but as the maxim goes: always use the smallest tool for the job.

There are a host of global factors to consider when extending the language, especially when it comes to introducing a new fundamental distinction. The payoff in this case seems incredibly tiny. And we *do* have plenty of first-hand experience with `?` traits in the form of `Sized`.

I wonder if you could spell out, in terms of *practical impact*, why you feel so strongly about built-in traits?

👍 2

Ericson2314 commented on 6 Apr • edited Contributor

I'll start a list.

1. Opt-out traits are *less* impactful for those that don't care. Don't care about weird FFI types? Never write `?DynSized`. If somebody else wants to use your crate for those, they can send the `DynSize` PR. C.F. with opt-in `DynSize` and deprecated `{size,align}_of_val`, now everyone needs to care *if* the new replacement methods are to get traction. This is the exact opposite of what **@withoutboats** said.

2. `DynSize` is a minor now, but seems like an important part of any custom DST proposal. Custom DSTs are very useful for things we care about.

3. Opt-out traits are like Cargo default features. They allow a completely different way of changing the library/language by reducing dependencies/assumptions instead of adding features. They are the *only* way to backwards compatibility do that we have, in fact.

   I am personally interested in this sort of thing. It's very similar to portability, for example. We want rust crates that don't or barely need a normal OS to also support weird platforms without annoying the crate author. It is an open "ecosystem sociology" question whether this can be pulled off. Similarly a bunch of us want truly unized types, custom DSTs, linear types, out pointers, and other weird things without pissing off regular uses. Opt-out traits, again, seem the *best and only* way to do that.

4. I strongly agree with whoever wrote that having more opt-out traits is *good* for pedagogy---it was a great point that I hadn't previously thought of at all. Right now `Sized`, being the one weird trait, isn't really part of a general pattern. DSTs, `Sized`, and opt-out traits are probably all one mess in most peoples head. Having more opt-out traits teases the concepts apart: who knows which opt-out trait you'll grok first, and now that can help you learn the others.

   I think it's illustrative that you wrote "built-in traits" above **@aturon**. We have many different types of magic traits today, from the most normal Copy (requires impl), to Send/Sync (implicit impl but not default bounds), and `Sized` (implicit impl and default bound). Making sure ever weird class has multiple examples and a dedicated names (better than old "OIBIT"! https://internals.rust-lang.org/t/pre-rfc-renaming-oibits-and-changing-their-declaration-syntax/3086/15) should clear things up.

cramertj commented on 6 Apr Member

> DynSize is a minor now, but seems like an important part of any custom DST proposal. Custom DSTs are very useful for things we care about.

@Ericson2314 I've yet to see any custom DST proposal that involved any types whose size was completely unknown at runtime. Why is this so critical?

Ericson2314 commented on 6 Apr • edited Contributor

@cramertj Custom DSTs are `DynSized` by definition. It's that implementing a trait is by far the most natural way to add the right hook. I want a repeat of `Copy: Clone` not `Drop`. `Drop` got it wrong because as all types (today) can be dropped, the question is when is the drop automatic and when does it require user code; I'd have preferred a `Forget: Destroy`.

Ericson2314 commented on 6 Apr • edited Contributor

@cramertj Also some custom types might have really expensive ways to calculate the size (C strings, for example). For performance-conscious users, it may be better to not implement `DynSized` and do all size look-ups by hand. IMO, all implicit operations being `O(1)` is a defensible if extreme position to take.

C.f. some people arguing similarly about lock guards being linear and needing to explicitly consume `unlock` in the past (not that i necessarily agree with that lock guard example).

Ericson2314 commented on 6 Apr Contributor

1. I don't think "always use the smallest tool for the job" applies here. The decreased power of lints is directly *worse* for uses. C.f. non-null lints v.s. `Option` in other languages. Lints are easily lost amid other warnings, and the fact is only some users will care. This means while individual code bases might obey them, the ecosystem as a whole can *not be trusted* to uphold the invariants the lints try to maintain. This a real loss for fostering an ecosystem of `DynSize` abstractions, or whatever the niche feature is, as for such niche things, being able to sync up few and scattered programmers and form a community is all the more important.
2. Ecosystem-wide enforcement is also good for the "regular users don't need to care" goal. If some library happens to use truly unsized types, and the consumer is unaware, they could face some nasty unexpected packages. With `?DynSize` they do get bothered with a compile time error they didn't expect, but that is much less nasty to deal with with than a run-time bug. If they don't want to learn `?DynSized`, they can go use a different library; better to have the opportunity to do that up front than after you're tied to the library too deeply because it took a while to excise the `{size,align}_of_val` panic.

👍 1

retep998 commented on 6 Apr • edited Member

As useful as extern types would be for me, having extern types without all the proper language machinery to enforce their unsizedness would result in a partial solution that is marginally better than the current partial solution of using zero variant enums. Extern types don't even solve the real problems for me such as the inability to properly specify that a struct is opaque beyond the first few fields or that a struct ends in dynamically sized or unsized data yet has a thin pointer. I want a full comprehensive plan for how to get to a full solution to those problems. What I don't want is any partial solution being stabilized early without being part of the full comprehensive plan, because that just leads to Rust being locked into something sub par.

👍 5 ❤️ 1

mikeyhew commented on 6 Apr Contributor

@aturon

> I wonder if you could spell out, in terms of practical impact, why you feel so strongly about built-in traits?

I want to create safe data structures for DSTs, like the DSTVec data structure that I posted about on Reddit a while ago (probably over a year ago), which stores DSTs contiguously in memory to avoid boxing. It requires `AlignFromMeta` and either `SizeFromMeta` or `SizeFromRef`, and I'd like to be able to write those requirements as a trait bound.

A few months ago, I came up with an idea that avoids the `?` altogether. I'm referring to the idea that if a `Sized`-family trait appears in the list of trait bounds, the default `Sized` bound is removed. I'd like to explore that by implementing it in tree and seeing what it's like to use it.

Like **@Ericson2314**, I don't think we want to pick the "smallest tool for the job" here, if "smallest" means the least powerful, least general, or least extensible. The Rust team has been pretty good about having a rigorous design process, and never just adding a language feature when the tools to implement it can be added instead, and the original feature possibly added as a syntactic sugar for something more expressive. In this case, the tools we are talking about are the `Sized`-family traits, and an `extern` type is really just a type that doesn't implement them.

👍 1 ❤️ 1

Ericson2314 commented on 7 Apr • edited Contributor

@mikeyhew Overall I very much agree with all that. One thing is though:

>

> I'm referring to the idea that if a `Sized`-family trait appears in the list of trait bounds, the default `Sized` bound is removed. I'd like to explore that by implementing it in tree and seeing what it's like to use it.

This would require us to design the entire hierarchy at once. Because otherwise, if we add another then now that one can't be disabled by the other older ones for backwards compatible. Better to have just one notion of `?`-traits than a courser staircase thing I think.

(BTW the one notion can be thought of as just one type of `?`-trait and a "flat" default bound `Size + DynSized + ...` such that any trait opted out also removes any other part of the default bound implying it.)

nikomatsakis commented on 10 Apr — Contributor

Note: **@joshtriplett** opened up #49708 to discuss the specific question of "what should the behavior of `size_of_val` (and `align_of_val`) be when applied to extern types" -- that is, the specific question at hand (which obviously intersects larger questions around `DynSized`).

(I didn't see him announce that here.)

👍 1

This comment was marked as off-topic. Show comment

This comment was marked as off-topic. Show comment

This comment was marked as off-topic. Show comment

This comment was marked as off-topic. Show comment

This comment was marked as off-topic. Show comment

bors added a commit that referenced this issue on 13 Apr

`Auto merge of #49669 - SimonSapin:global-alloc, r=alexcrichton` ··· ✓ `99d4886`