

Compiler Driven Development

Making Rust Work for You

Jacob Pratt
2024-09-12
Montréal



RUSTCONF
2024

Diagnostics

- overview

Diagnostics

- overview
- warnings vs. errors

Diagnostics

- overview
- warnings vs. errors

	warning	error
fatal	no	yes

Diagnostics

- overview
- warnings vs. errors

	warning	error
fatal	no	yes
issue	potential	near- certain

Diagnostics

- overview
- warnings vs. errors

	warning	error
fatal	no	yes
issue	potential	near- certain
action	<i>should</i> be taken	<i>must</i> be taken

Diagnostics

- overview
- warnings vs. errors

Diagnostics

- overview
- warnings vs. errors
- reading diagnostics: *do it!*

Diagnostics

- overview
- warnings vs. errors
- reading diagnostics: *do it!*

```
#[derive(Default)]  
enum Foo {  
    #[default]  
    #[default]  
    Alpha,  
    Beta,  
}
```

Diagnostics

- overview
- warnings vs. errors
- reading diagnostics: *do it!*

```
#[derive(Default)]
enum Foo {
    #[default]
    #[default]
    Alpha,
    Beta,
}

error: multiple `#[default]` attributes
--> src/lib.rs:5:5
3 |     #[default]
  |     ----- `#[default]` used here
4 |     #[default]
  |     ----- `#[default]` used again here
5 |     Alpha,
  |     ^^^^^
= note: only one `#[default]` attribute is needed
help: try removing this
--> src/lib.rs:4:5
4 |     #[default]
  |     ^^^^^^^^^
```

Lints

Lints

- why?

Lints

- why?
 - catch bugs

Lints

- why?
 - catch bugs
 - improve performance

Lints

- why?
 - catch bugs
 - improve performance
 - write idiomatic code

Lints

- why?
 - catch bugs
 - improve performance
 - write idiomatic code
- built-in vs. clippy

Lints

- why?
 - catch bugs
 - improve performance
 - write idiomatic code
- built-in vs. clippy
- which should be enabled

Lints

- why?
 - catch bugs
 - improve performance
 - write idiomatic code
- built-in vs. clippy
- which should be enabled
- enabling as warn vs. deny

Lints

- why?
 - catch bugs
 - improve performance
 - write idiomatic code
- built-in vs. clippy
- which should be enabled
- enabling as warn vs. deny
 - defaults are a great starting place

Lints

- why?
 - catch bugs
 - improve performance
 - write idiomatic code
- built-in vs. clippy
- which should be enabled
- enabling as warn vs. deny
 - defaults are a great starting place
 - should not *decrease* severity without strong reason

Rust Analyzer

Rust Analyzer

- check on save

Rust Analyzer

- check on save
- clippy

Rust Analyzer

- check on save
- clippy
- inlay hints

```
0 implementations | 1 reference
1 struct X {
2     a: u8,    fields `a`, `b`, and `c` are never read
3     b: u8,    fields `a`, `b`, and `c` are never read
4     c: u8,    fields `a`, `b`, and `c` are never read
5 }
6
7 ▶ Run | Debug | 0 references
7 fn main() {
8     let x: X = X { a: 0, b: 0, c: 0 };    unused variable: `x`
9 }
10
```


Rust Analyzer

- check on save
- clippy
- inlay hints
- code completion

```
0 implementations | 1 reference
1 struct X {
2     a: u8,    fields `a`, `b`, and `c` are never read
3     b: u8,    fields `a`, `b`, and `c` are never read
4     c: u8,    fields `a`, `b`, and `c` are never read
5 }
6
0 references | ▶ Run | Debug
7 fn main() {
8     let x: X = X { a: 0, b: 0, c: 0 };    unused variable: `x`
9     x.a = 1;    Syntax Error: expected field name or number
10 }
11
```

☒ a u8

☒ b u8

☒ c u8

☐ arc (use std::sync::A... Put the expression into...

☐ box Box::new(expr)

☐ call function(expr)

☐ dbg dbg!(expr)

☐ dbgr dbg!(&expr)

☐ deref *expr

☐ err Wrap the expression in a `Result::Err`

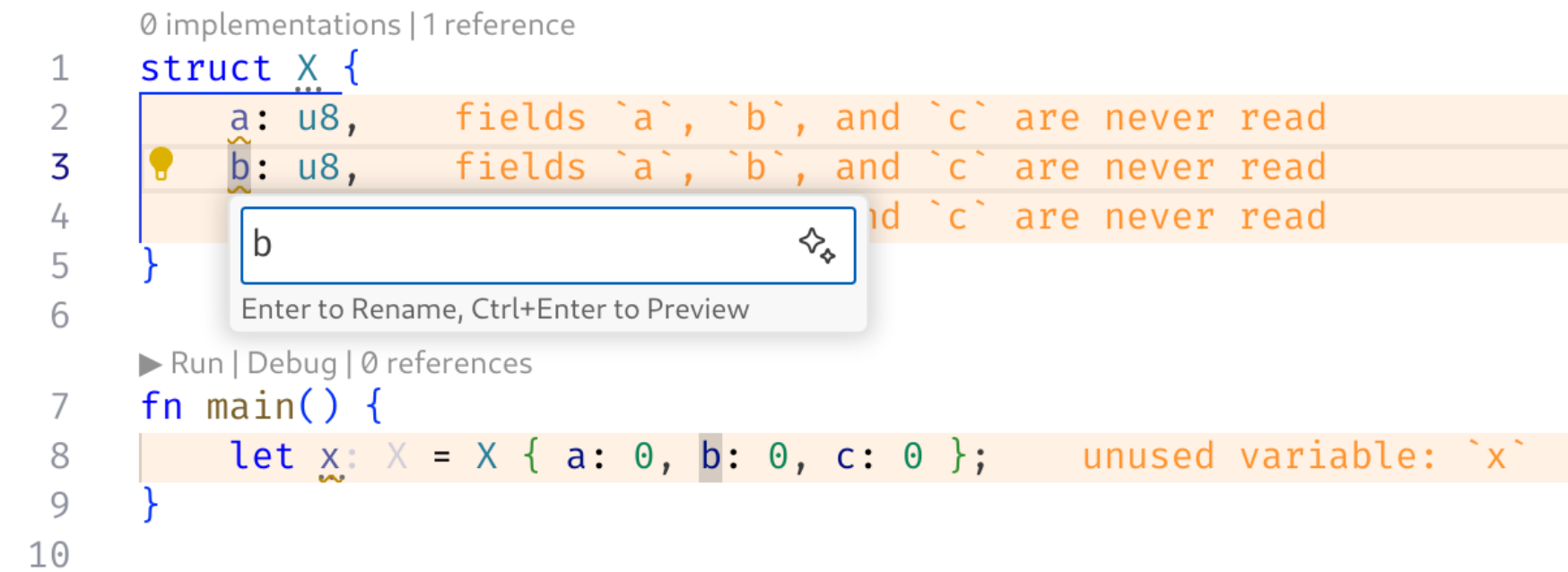
☒ into() (as Into) fn(self) → T

☐ let let

Rust Analyzer

- check on save
- clippy
- inlay hints
- code completion
- refactoring tools

```
1 0 implementations | 1 reference
2 struct X {
3     a: u8, fields `a`, `b`, and `c` are never read
4     b: u8, fields `a`, `b`, and `c` are never read
5 }
6
7 ► Run | Debug | 0 references
8 fn main() {
9     let x: X = X { a: 0, b: 0, c: 0 }; unused variable: `x`
10 }
```

A screenshot of the Rust Analyzer interface. The top part shows a struct definition: `struct X { a: u8, b: u8 }`. The variable `b` is being typed, and a completion popup shows the suggestion `b`. Below the struct, a function `main` is defined with a line `let x: X = X { a: 0, b: 0, c: 0 };`. The variable `x` is underlined with a wavy line, and a diagnostic message `unused variable: `x`` is shown. The interface also displays inlay hints for the struct fields: `fields `a`, `b`, and `c` are never read`. The bottom part of the interface shows the `Run` and `Debug` buttons, and the number of references for the selected variable.

Rust Analyzer

- check on save
- clippy
- inlay hints
- code completion
- refactoring tools
- code navigation

The screenshot displays the Rust Analyzer interface. At the top, a search bar shows '0 implementations | 1 reference' for the symbol 'X'. Below this, the main editor window shows the file 'main.rs' at path '~/.tmp/src'. The code in the editor is as follows:

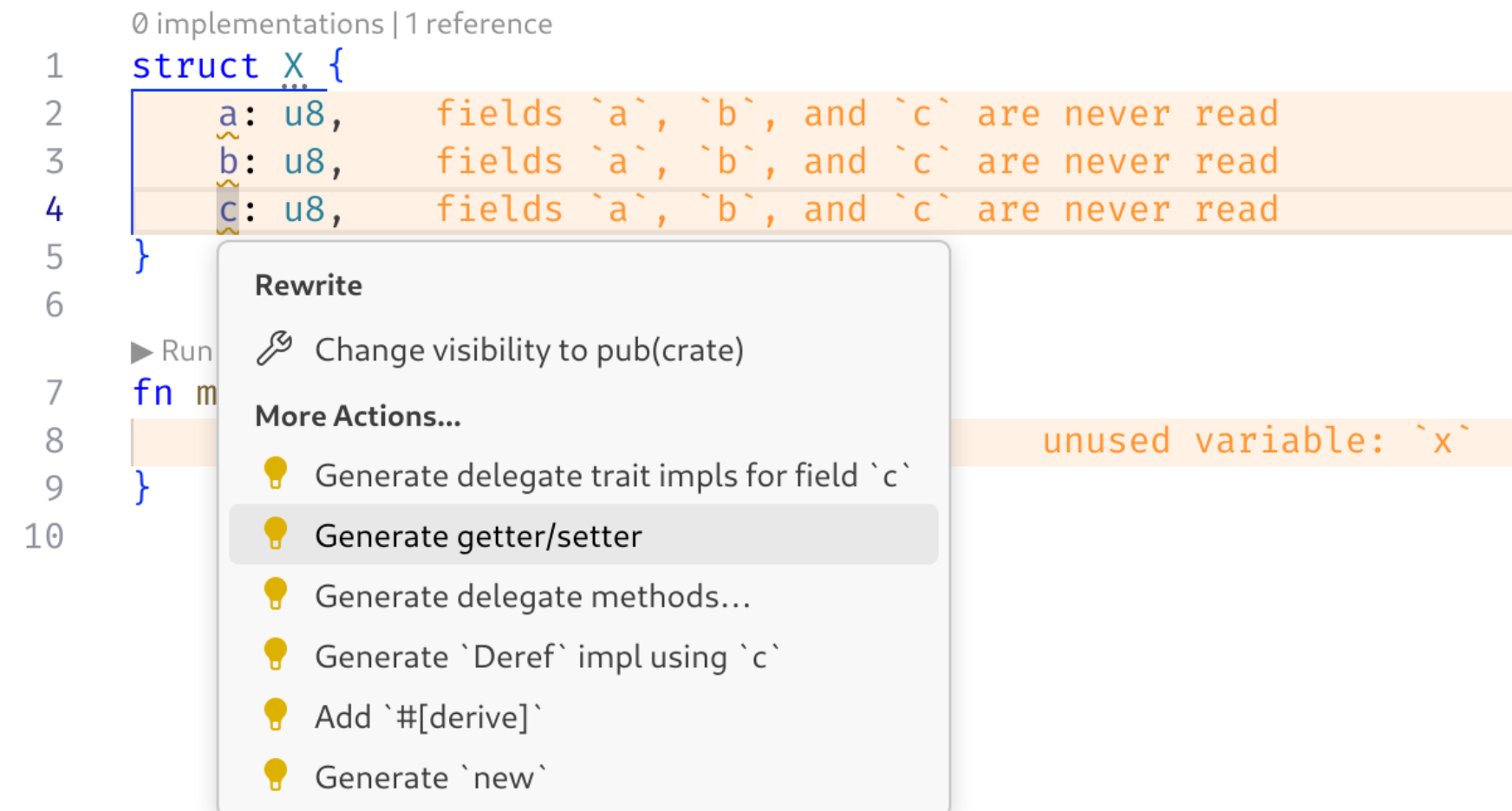
```
1 struct X {  
2     a: u8,  
3     b: u8,  
4     c: u8,  
5 }  
6  
7 fn main() {  
8     let x: X = X { a: 0, b: 0, c: 0 };  
9 }  
10
```

On the right side of the editor, a code completion popup is visible, showing the suggestion 'let x = X { a: 0, b: 0, c: 0 };'. Below the editor, a panel displays inlay hints for the struct fields. For each field, it indicates that the fields 'a', 'b', and 'c' are never read. Additionally, a hint for the variable 'x' in the main function indicates it is an 'unused variable'.

```
2 a: u8, fields `a`, `b`, and `c` are never read  
3 b: u8, fields `a`, `b`, and `c` are never read  
4 c: u8, fields `a`, `b`, and `c` are never read  
5 }  
6  
7 Run | Debug | 0 references  
8 fn main() {  
9     let x: X = X { a: 0, b: 0, c: 0 }; unused variable: `x`  
10 }
```

Rust Analyzer

- check on save
- clippy
- inlay hints
- code completion
- refactoring tools
- code navigation
- basic code generation



Type system

Type system

- "if it compiles, it works"

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
 - null pointer exceptions (no `null`)

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
 - null pointer exceptions (no `null`)
 - use-after-free (lifetimes)

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
 - null pointer exceptions (no `null`)
 - use-after-free (lifetimes)
 - dangling pointers (lifetimes)

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
 - null pointer exceptions (no `null`)
 - use-after-free (lifetimes)
 - dangling pointers (lifetimes)
 - double frees (lifetimes)

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
 - null pointer exceptions (no `null`)
 - use-after-free (lifetimes)
 - dangling pointers (lifetimes)
 - double frees (lifetimes)
 - data races (exclusive mutability)

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
 - null pointer exceptions (no `null`)
 - use-after-free (lifetimes)
 - dangling pointers (lifetimes)
 - double frees (lifetimes)
 - data races (exclusive mutability)
 - uninitialized reads (type system)

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
- no need for humans to check many things

Type system

- "if it compiles, it works"
- "fighting the borrow checker" is a feature
- entire classes of bugs eliminated
- no need for humans to check many things
- large refactorings are acceptable

Design patterns

Design patterns

- newtype pattern

Design patterns

- newtype pattern
 - extend and/or restrict functionality of the wrapped type

Design patterns

- newtype pattern
 - extend and/or restrict functionality of the wrapped type
 - enforce different types for different uses

Design patterns

- newtype pattern
 - extend and/or restrict functionality of the wrapped type
 - enforce different types for different uses

```
struct UserId(u64);
```

Design patterns

- newtype pattern
 - extend and/or restrict functionality of the wrapped type
 - enforce different types for different uses

Design patterns

- newtype pattern
 - extend and/or restrict functionality of the wrapped type
 - enforce different types for different uses
 - avoid leaking sensitive information

Design patterns

- newtype pattern
 - extend and/or restrict functionality of the wrapped type
 - enforce different types for different uses
 - avoid leaking sensitive information

```
struct Password(String);  
impl Debug for Password {  
    fn fmt(&self, f: &mut Formatter) -> Result {  
        f.write_str("Password(****)")  
    }  
}
```

Design patterns

- newtype pattern

Design patterns

- newtype pattern
- builder pattern

```
let foo = Foo
    .alpha()
    .beta(0)
    .gamma("hello")
    .build();
```

Design patterns

- newtype pattern
- builder pattern

Design patterns

- newtype pattern
- builder pattern
- typestate

```
struct Drone<State>(/* ... */);  
struct Landed;  
struct Hovering;
```

```
impl Drone<Landed> {  
    fn take_off(self) -> Drone<Hovering> { /* ... */ }  
}  
impl Drone<Hovering> {  
    fn land(self) -> Drone<Landed> { /* ... */ }  
}
```

```
Drone::<Landed>(/* ... */).land(); // compile error
```


Pitfalls

Pitfalls

- `unwrap()` and `expect()`

Pitfalls

- `unwrap()` and `expect()`
- lifetime virality

Pitfalls

- `unwrap()` and `expect()`
- lifetime virality
- `unsafe` code

Pitfalls

- `unwrap()` and `expect()`
- lifetime virality
- `unsafe` code
- usability

Pitfalls

- `unwrap()` and `expect()`
- lifetime virality
- `unsafe` code
- usability
 - keep it simple!

Example: provenance

Example: provenance

```
trait ByteParser<'input> {  
    /* ... */  
    fn parse_bytes<'prov>(self, input: Bytes<'input, 'prov>)  
        -> IntermediateResult<'input, 'prov, Self::Output, Self::Error>;  
}
```


Example: provenance

```
trait ByteParser<'input> {  
    /* ... */  
    fn parse_bytes<'prov>(self, input: Bytes<'input, 'prov>)  
        -> IntermediateResult<'input, 'prov, Self::Output, Self::Error>;  
}  
  
struct Bytes<'input, 'prov> {  
    bytes: &'input [u8],  
    variance: PhantomData<fn(&'prov ()) -> &'prov ()>,  
}
```

Example: provenance

```
trait ByteParser<'input> {  
    /* ... */  
    fn parse_bytes<'prov>(self, input: Bytes<'input, 'prov>)  
        -> IntermediateResult<'input, 'prov, Self::Output, Self::Error>;  
}
```

Example: provenance

```
trait ByteParser<'input> {  
    /* ... */  
    fn parse_bytes<'prov>(self, input: Bytes<'input, 'prov>)  
        -> IntermediateResult<'input, 'prov, Self::Output, Self::Error>;  
}  
  
impl Bytes<'input, 'prov> {  
    unsafe fn from_slice(bytes: &'input [u8]) -> Self { /* ... */ }  
    fn split_at(&self, mid: usize)  
        -> Result<(&'input [u8], Self), EndOfInput> { /* ... */ }  
}
```


Example: provenance

```
trait ByteParser<'input> {  
    /* ... */  
    fn parse_bytes<'prov>(self, input: Bytes<'input, 'prov>)  
        -> IntermediateResult<'input, 'prov, Self::Output, Self::Error>;  
}  
  
impl Bytes<'input, 'prov> {  
    unsafe fn from_slice(bytes: &'input [u8]) -> Self { /* ... */ }  
    fn split_at(&self, mid: usize)  
        -> Result<(&'input [u8], Self), EndOfInput> { /* ... */ }  
}  
  
fn parse<'input, R>(  
    input: &'input [u8],  
    f: impl for<'prov> FnOnce(Bytes<'input, 'prov>) -> R,  
) -> R {  
    f(unsafe { Bytes::from_slice(input.as_slice()) })  
}
```

Compiler Driven Development

Making Rust Work for You



Jacob Pratt



@jhpratt



@jhpratt@mastodon.social



jhpratt.dev



sponsor.jhpratt.dev

2024-09-12
Montréal



RUSTCONF
2024