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	*should* be taken	*must* 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

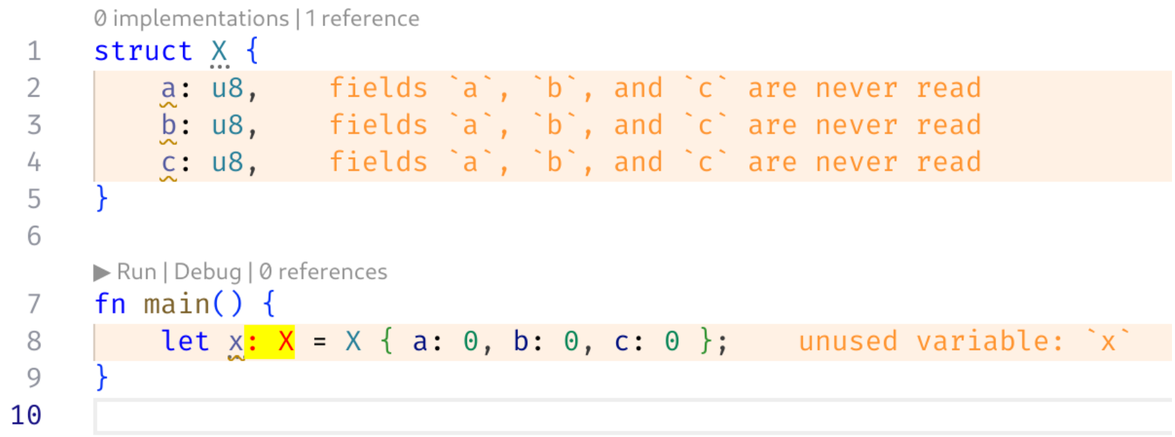
- check on save

Rust Analyzer

- check on save
- clippy

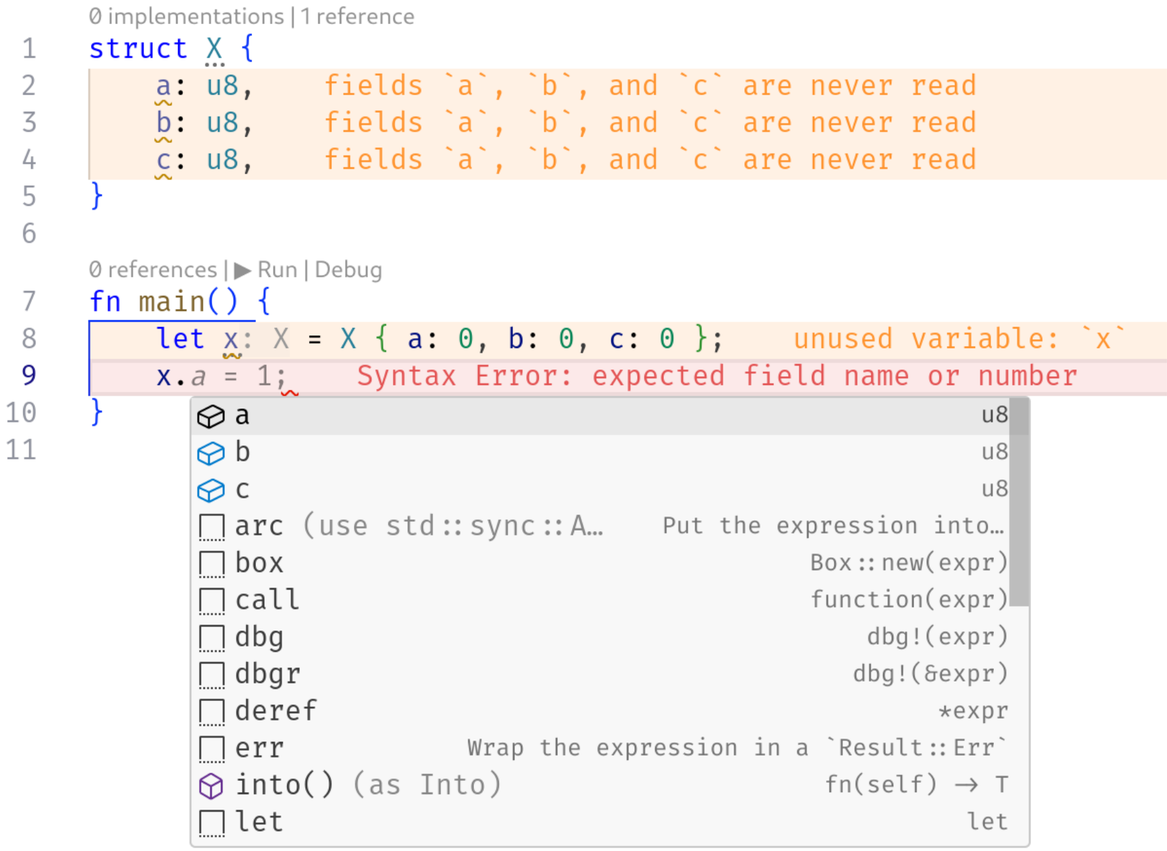
Rust Analyzer

- check on save
- clippy
- inlay hints



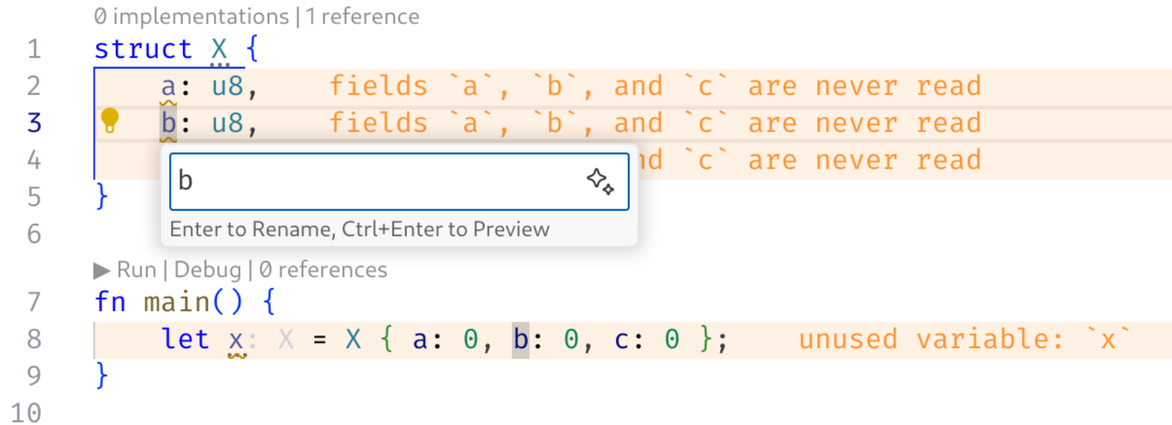
Rust Analyzer

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



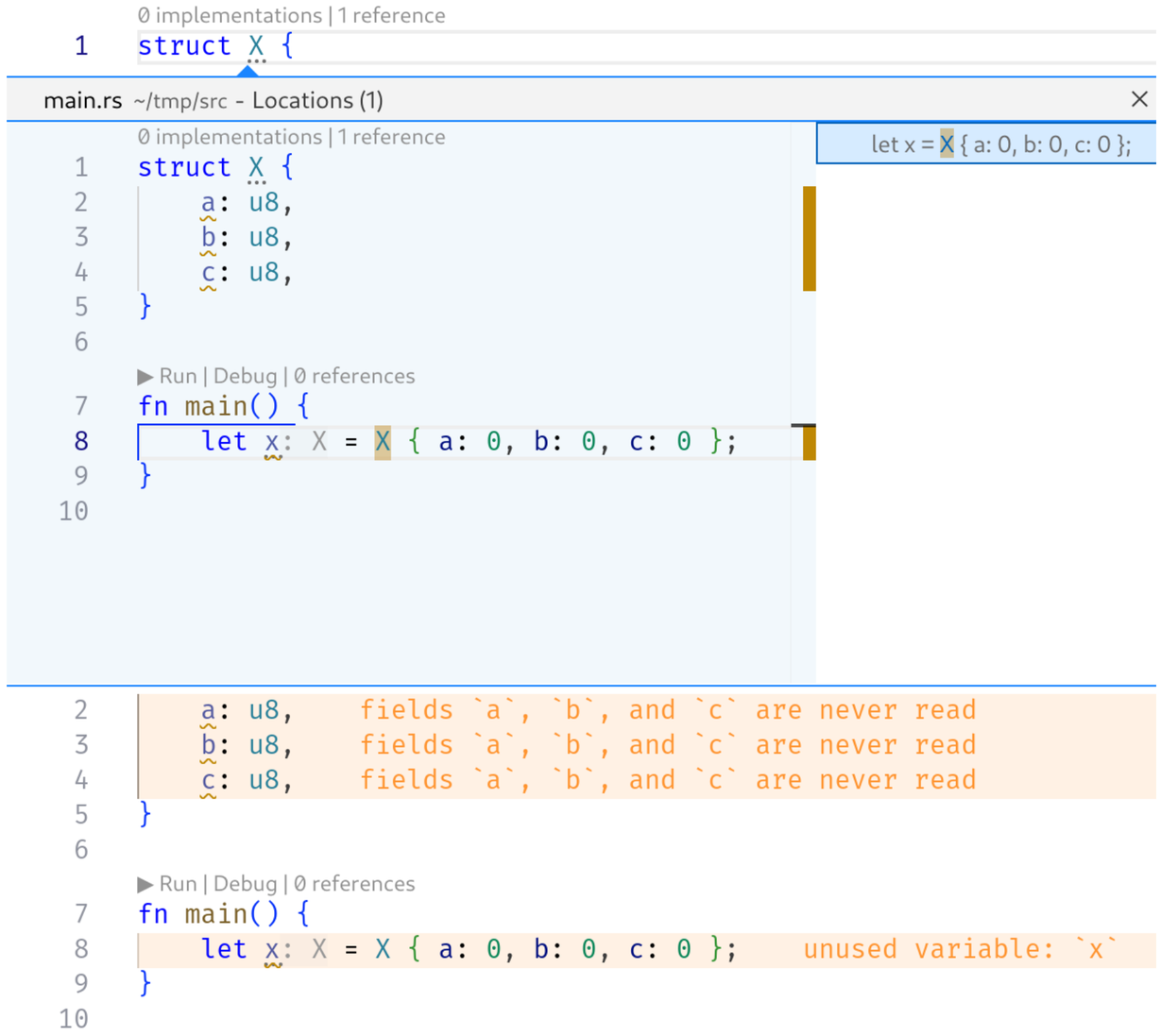
Rust Analyzer

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



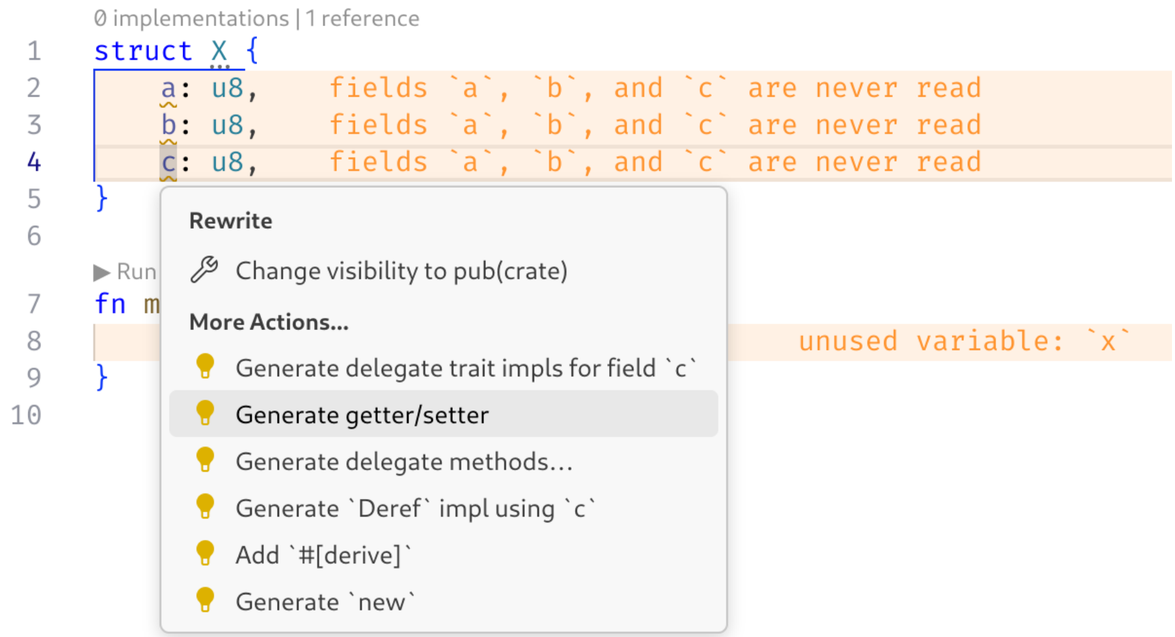
Rust Analyzer

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



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