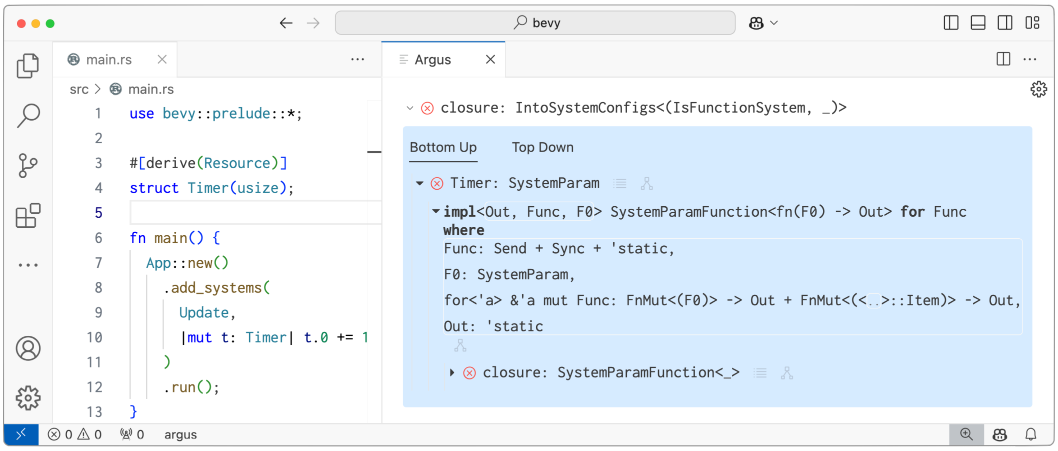
An Interactive Debugger for Rust Trait Errors

GAVIN GRAY, WILL CRICHTON, and SHRIRAM KRISHNAMURTHI, Brown University, USA



```
use bevy::prelude::*;

#[derive(Resource)]
struct Timer(usize);

fn main() {
  App::new()
    .add_systems(
      Update,
      |mut t: Timer| t.0 += 1
    )
    .run();
}
```

```
closure: IntoSystemConfigs<(IsFunctionSystem, _)>
  Bottom Up    Top Down
  Timer: SystemParam
    impl<Out, Func, F0> SystemParamFunction<fn(F0) -> Out> for Func
    where
    Func: Send + Sync + 'static,
    F0: SystemParam,
    for<'a> &'a mut Func: FnMut<(F0)> -> Out + FnMut<(<..>::Item)> -> Out,
    Out: 'static
      closure: SystemParamFunction<_>
```

Fig. 1. A screenshot of the Argus trait debugger's bottom-up view in VS Code applied to a Bevy program (Section 2.3). Using an interactive graphical interface for the trait inference tree, Argus can include key information (the bound `Timer: SystemParam`) elided by the Rust compiler diagnostic for the same program.

Compiler diagnostics for type inference failures are notoriously bad, and type classes only make the problem worse. By introducing a complex search process during inference, type classes can lead to wholly inscrutable or useless errors. We describe a system, Argus, for interactively visualizing type class inferences to help programmers debug inference failures, applied specifically to Rust's trait system. The core insight of Argus is to avoid the traditional model of compiler diagnostics as one-size-fits-all, instead providing the programmer with different views on the search tree corresponding to different debugging goals. Argus carefully uses defaults to improve debugging productivity, including interface design (e.g., not showing full paths of types by default) and heuristics (e.g., sorting obligations based on the expected complexity of fixing them). We evaluated Argus in a user study where $N = 25$ participants debugged type inference failures in realistic Rust programs, finding that participants using Argus correctly localized 2.2× as many faults and localized 3.3× faster compared to not using Argus.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Constraint and logic programming**; • **Information systems** → **Search interfaces**.

Additional Key Words and Phrases: Rust, type classes, traits, debugging

ACM Reference Format:
Gavin Gray, Will Crichton, and Shriram Krishnamurthi. 2025. An Interactive Debugger for Rust Trait Errors. *Proc. ACM Program. Lang.* 9, PLDI, Article 199 (June 2025), 22 pages. https://doi.org/10.1145/3729302

Authors' Contact Information: Gavin Gray, gavin_gray@brown.edu; Will Crichton, will_crichton@brown.edu; Shriram Krishnamurthi, Brown University, Providence, USA.

ACM 2475-1421/2025/6-ART199
https://doi.org/10.1145/3729302

1 Introduction

Type classes improve the brevity of bounded polymorphism by implicitly passing inferred type class instances to generic functions, as opposed to ML-style explicit passing of modules via functors. In other words, type classes shift the burden of finding instances from the programmer to the compiler. When a type class inference succeeds, the programmer does not need to expend any thought on the inference process, at least for systems which enforce coherence [9, 22]. But when a type class inference fails, the compiler is responsible for explaining the failure to the programmer.

Back in the days when type classes were a simple tool for overloading [30], diagnostics posed no particular challenge. Say a function has a type like `ToString a => a -> String`, and you pass an `int`, and there happens to be no instance of `ToString` for `int`. Then the compiler just needs to say: "no instance found for `ToString int`".

The challenge today is that type class systems are more powerful than before. Type classes in Haskell, Coq, Rust (traits), and Scala (implicits) have all been shown to encode Turing-complete computations. Rust even explicitly models type class inference ("trait solving") as Prolog-esque logic programming [4]. Library developers in these languages, especially Rust, increasingly lean on type classes to encode domain-specific correctness properties into the type system. While this approach helps developers catch more mistakes at compile-time, it can at times produce mystifying diagnostics. Given the severity of this problem, the Rust community has invested money into studying trait diagnostics [25] as well as developed both library-specific [18] and library-agnostic [26] utilities solely for improving trait diagnostics.

The thesis of this work is that compiler diagnostics are fundamentally limited by their representation as static text. Moreover, this limitation is felt most acutely for information-rich situations such as type class inference. We therefore designed a system, Argus, to provide a richer interface for explaining type class inferences built on a modern UI framework. Argus is implemented as an IDE extension for Rust, although its core design is not particularly Rust-specific. After motivating our design principles with concrete examples (Section 2), we describe our contributions:

- A novel interface for visualizing trait inference, designed to specifically facilitate key sub-tasks in debugging inference failures (Section 3.2).
- A new heuristic, *inertia*, that ranks potential root causes of trait inference failure (Section 3.3).
- A user study that shows that participants using Argus could localize faults 3.3× faster compared to using the Rust compiler's diagnostics (Section 5).

2 Motivating Examples

Traits are a well-documented source of confusing compiler errors in the Rust community. A 2023 study [25] commissioned by the Rust Foundation identified dozens of problematic error messages in widely-used libraries, resulting in a corpus of hard-to-debug programs. We analyzed the content of these error messages to form hypotheses about how to design better trait diagnostics.

We chose three programs that represent the main failure modes of programs in this corpus. For illustration, the program in Section 2.2 was taken from an online Rust forum [1] because it requires less boilerplate than programs in the corpus, but nonetheless contains an equivalent error. We start by walking through these concrete examples that illustrate the problems in Rust's existing trait diagnostics. We then generalize these examples into design principles that form the basis of the Argus interface. All errors in this section were generated using Rust 1.82.0.

2.1 A Missing Table Join

Diesel [3] is a popular Rust library for object-relational mapping and statically-checked query building. Figure 2a shows an example Diesel program where a developer wants to select fields from

```
fn users_with_eq_post_id(conn: &mut PgConnection) -> Vec<(i32, String)> {
  users::table // .inner_join(posts::table)
    .filter(users::id.eq(posts::id))
    .select((users::id, posts::id))
    .load::<(i32, String)>(conn)
}
```

(a) A program using the Diesel query builder library. The program does not join the table `posts` but tries to use the `posts::id` column, which Diesel catches as a trait error.

```
error[E0271]: type mismatch resolving `<table as AppearsInFromClause<table>>::Count == Once`
     |
 5   |         .load::<(i32, String)>(conn);
     |          ----                 ^^^^ expected `Once`, found `Never`
     |          |
     |          required by a bound introduced by this call
     |
note: required for `posts::columns::id` to implement `AppearsOnTable<users::table>`
     |
18   |         id -> Integer,
     |         ^^
     = note: associated types for the current `impl` cannot be restricted in `where` clauses
     =  note: 2 redundant requirements hidden
     = note: required for `diesel::expression::grouped::Grouped<diesel::expression::operators::Eq<users::
      columns::id, posts::columns::id>>` to implement `AppearsOnTable<users::table>`
     = note: required for `query_builder::where_clause::WhereClause<diesel::expression::grouped::Grouped<
      diesel::expression::operators::Eq<users::columns::id, posts::columns::id>>>` to implement `
      query_builder::where_clause::ValidWhereClause<FromClause<users::table>>`
     = note: required for `SelectStatement<FromClause<table>, SelectClause<(id, name)>, NoDistinctClause,
      WhereClause<Grouped<Eq<id, id>>>>` to implement `Query`
     = note: required for `SelectStatement<FromClause<table>, SelectClause<(id, name)>, NoDistinctClause,
      WhereClause<Grouped<Eq<id, id>>>>` to implement `LoadQuery<'_, _, (i32, String)>`
note: required by a bound in `diesel::RunQueryDsl::load`
     |
1540 |     fn load<'query, U>(self, conn: &mut Conn) -> QueryResult<Vec<U>>
     |        ---- required by a bound in this associated function
1541 |     where
1542 |         Self: LoadQuery<'query, Conn, U>,
     |               ^^^^^^^^^^^^^^^^^^^^^^^^^^ required by this bound in `RunQueryDsl::load`
```

(b) The Rust compiler diagnostic for the program above.

Fig. 2. An example of a trait error where the compiler elides key information for brevity. This is noted by the phrase "2 redundant requirements hidden."

two tables, `users` and `posts`, but forgot to join the `posts` table into the query. Diesel uses traits to discover that the call to `.load(conn)` is ill-typed because the query selects a field of a missing table, generating the diagnostic shown in Figure 2b. To debug this trait error, a developer must *localize* the root cause (i.e., the `.eq(post::id)` operation) and *fix* the program (e.g., by inserting a join).

The diagnostic's goal is principally to help with the localization phase of debugging by providing context about the origin of the type error. For instance, the Rust compiler diagnostic in Figure 2b does not just report the top-level failed trait bound, printed at the very bottom of the diagnostic. Rust instead starts by reporting the failed predicate deepest into the trait inference tree:

```
type mismatch resolving `<table as AppearsInFromClause<table>>::Count == Once`
```

The developer's localization task is to blame a specific program element for this failed predicate. This task presents two problems. First, the associated type `AppearsInFromClause::Count` may not be self-evidently meaningful, requiring additional context to interpret the constraint (e.g., where did this constraint come from?). Second, the types `users::table` and `posts::table` have been unfortunately truncated to simply `table`, suggesting the types are the same when they are not.

To try solving problem #1, the developer could read the rest of the diagnostic. The remaining text explains the provenance of the constraint, which is a sequence of five trait constraints deriving from the originating constraint on the call to `.load(conn)`. One possibly useful constraint to read would be `Eq<users::columns::id, posts::columns::id>: AppearsOnTable<users::table>`. The `Eq<...>` type hints at the problem originating with the expression `users::id.eq(posts::id)`. This bound helps the developer localize the fault: that `posts` must be joined before using this expression.

However, this constraint does not actually appear in the text of the diagnostic! It is elided with the statement: `= note: 2 redundant requirements hidden`. Also observe that the diagnostic includes the source code and location for the first two trait bounds walking up from the deepest failed bound (at the top of the diagnostic) and the originating trait bound (at the bottom of the diagnostic). The diagnostic does not include this information for any of the intermediate trait bounds.

The Rust compiler omits all this information out of necessity, not convenience. Consider the counterfactual where Rust includes the full text of every bound and its source-mapped origin. This diagnostic could easily stretch over 100 lines long, just for a relatively simple error. Therefore, Rust applies heuristics to include only information that is probably relevant. The problem with identical-looking `table` types is similar. Rust heuristically decides when to present fully-qualified versus shortened paths for brevity, but it sometimes makes a wrong decision. Without representing diagnostics as static text, we can consider alternative solutions to both problems:

Principle CollapseSeq. Instead of omitting steps of an inference sequence for brevity, allow the developer to progressively unfold the sequence.

Principle ShortTys. Instead of heuristically shortening types, show shortened types by default, but make fully-qualified types available on-demand.

2.2 An Accidental Infinite Recursion

A Rust developer was designing an AST data type to be generic with respect to user-specific node-associated data. They wrote the code in Figure 3a, which caused an infinite loop in the trait solver, as indicated in the trait diagnostic in Figure 3b. The developer asked on a Rust forum [1]:

> I'm running into a compiler error, stating that there is an overflow when evaluating a trait requirement. However, that requirement should obviously be satisfied. I just can't seem to understand where the overflow comes from.

The actual loop has a simple logical structure, as shown in Figure 3c. If `EmptyNode` needs to implement `AstAssocs` (due to line 18), then that requires `EmptyNode` implements `AssocData<EmptyNode>` (due to line 10), which in turn requires `EmptyNode` implements `AstAssocs` (due to line 15).

However, the Rust diagnostic obscures this fact because the diagnostic interleaves the "core" information used in the trait solver (the trait-bounds and impl blocks) with "auxiliary" information used for debugging (the source-location of constraints). This approach makes it harder for a developer to identify the logical structure of the cycle. Again, the ultimate issue is the static text representation, which requires diagnostics to commit to a specific sequential interleaving of all relevant information. Therefore our principle is:

Principle CtxtLinks. Instead of interleaving the trait inference steps with auxiliary information, enable developers to access auxiliary information on-demand through contextual links.

```
1  trait AssocData<A: AstAssocs> {}
2  trait AstAssocs: Sized {
3    type Data: AssocData<Self>;
4  }
5
6  struct EmptyNode;
7  struct Statement<A: AstAssocs>(..);
8
9  impl<Data> AstAssocs for Data
10 where Data: AssocData<Self> {
11   type Data = Data;
12 }
13
14 impl<A> AssocData<A> for EmptyNode
15 where A: AstAssocs {}
16
17 fn main() {
18   let s: Statement<EmptyNode> =
19     Statement(..);
20 }
```

(a) A program which tries to model an AST with user-specified associated data on AST nodes. The described trait bounds and impl blocks cause an infinite loop in the trait solver.

```
error[E0275]: overflow evaluating the requirement `
     EmptyNode: AssocData<EmptyNode>`
   |
18 |   let s: Statement<EmptyNode> =
   |          ^^^^^^^^^^^^^^^^^^^^
   |
note: required for `EmptyNode` to implement `AstAssocs`
   |
 9 | impl<Data> AstAssocs for Data
   |            ^^^^^^^^^     ^^^^
10 | where Data: AssocData<Self> {
   |             ---------------
       unsatisfied trait bound introduced here
note: required by a bound in `Statement`
   |
 7 | struct Statement<A: AstAssocs>(
   |                     ^^^^^^^^^
          required by this bound in `Statement`
```

(b) The Rust compiler diagnostic for the program on the left.

$$\begin{array}{rl} & \texttt{EmptyNode: AstAssocs} \\ \xLeftarrow{\texttt{impl 9-12}} & \texttt{EmptyNode: AssocData<EmptyNode>} \\ \xLeftarrow{\texttt{impl 14-15}} & \texttt{EmptyNode: AstAssocs} \end{array}$$

(c) A diagrammatic representation of the logical structure of the recursion.

Fig. 3. An example of a trait error where the interleaving of information in the diagnostic obscures the logical structure of the problem.

2.3 An Errant Function Parameter

Bevy [2] is a popular Rust library for writing 2D and 3D games in the entity-component-system (ECS) style. Systems in ECS are functions that perform updates on the game. A system's function parameters declare the required inputs to the system, and the game engine essentially does dependency injection to run the system with the appropriate inputs.

For example, Figure 4a shows a developer trying to write a system which increments a global mutable timer. The correct approach is to declare `Timer` as a `Resource`, and then to use the container type `ResMut<Timer>` as the function parameter. However, a common Bevy mistake is to forget the container and simply write `timer: Timer`. The function `run_timer` is still well-typed, but now Bevy rejects the developer's attempt to register the system on the game. The type error arises from a failed trait inference, where the method `add_systems` requires that `run_timer` implements a trait `IntoSystem`, which converts a value into a system. A function system requires each parameter to implement a trait `SystemParam`, for which one implementation is that `ResMut<T>: SystemParam` if `T: Resource`. `Timer` does not implement `SystemParam`, so `run_timer` does not implement `IntoSystem`.

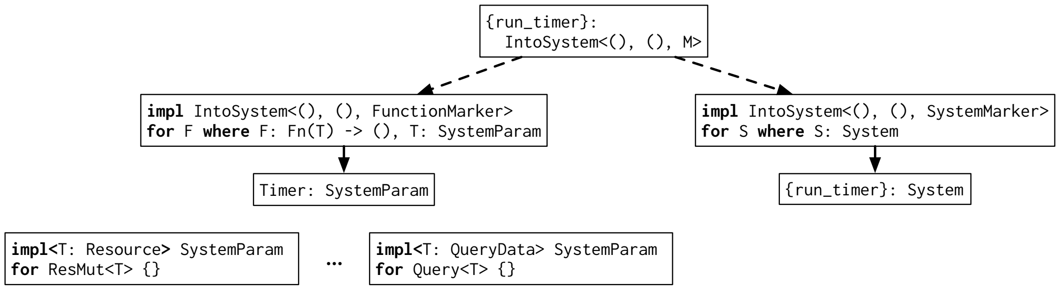
The problem is that the Rust diagnostic, shown in Figure 4b, only mentions the `IntoSystem` bound. It says, essentially, "something is wrong with the type of `run_timer`" without pointing to a more specific culprit. Rust lacks specificity because there are other ways to potentially implement

```
#[derive(Resource)]
struct Timer(usize);

fn run_timer(
//mut timer: ResMut<Timer>
  mut timer: Timer
) { timer.0 += 1; }

fn main() {
  App::new()
    .insert_resource(Timer(0))
    .add_systems(Update, run_timer)
    .run();
}
```

(a) A program using the Bevy game engine. The `run_timer` function incorrectly takes a parameter of type `Timer` instead of `ResMut<Timer>`.

```
error[E0277]: `fn(Timer) {run_timer}` does not
     describe a valid system configuration
     |
12   |   .add_systems(Update, run_timer)
     |    -----------         ^^^^^^^^^
     |    |          invalid system configuration
     |    |
     |  required by a bound introduced by this call
     |
= help: the trait `IntoSystem<(), (), _>` is not
     implemented for fn item `fn(Timer) {run_timer
     }`, which is required by `fn(Timer) {run_timer
     }: IntoSystemConfigs<_>`
```

(b) The Rust compiler diagnostic for the program on the left.


{run_timer}:
IntoSystem<(), (), M>
impl IntoSystem<(), (), FunctionMarker>
for F where F: Fn(T) -> (), T: SystemParam
impl IntoSystem<(), (), SystemMarker>
for S where S: System
Timer: SystemParam
{run_timer}: System
impl<T: Resource> SystemParam
for ResMut<T> {}
...
impl<T: QueryData> SystemParam
for Query<T> {}

(c) A diagrammatic representation of the relevant fragment of the trait inference tree, showing the branch point in the possible implementations of the `IntoSystem` trait.

Fig. 4. An example of a trait error where a branch point in the inference process causes the diagnostic to omit information deeper in the search tree.

`IntoSystem` for `run_timer`. The diagram in Figure 4c shows how `IntoSystem` can also be implemented for types that implement the `System` trait.[1]

Once more, we observe the limitations of the static text representation. Rust adopts the approach that when a branch point exists in the trait inference tree, its diagnostics stop at the branch point and do not provide finer-grained details along every branch. In other words, the diagnostics are constrained to presenting a *sequence* of information, not a *tree* of information. Therefore, we adopt the principle:

Principle TreeData. Instead of omitting tree-shaped information in a trait inference, provide an interface that supports exploring trait inference as a tree.

[1]These implementations seems to violate coherence. Indeed, with a straightforward definition of `IntoSystem`, Rust would reject the two impl blocks as overlapping. Bevy employs the technique of adding a *marker type parameter* to the `IntoSystem` trait, written as `FunctionMarker` and `SystemMarker` in Figure 4c. This parameter ensures that the two implementations are not strictly overlapping. It then increases the burden on Rust's type inference to deduce the correct type of the marker.

$$\text{Type Variable } \alpha \quad \text{Region Variable } \varrho \quad \text{Type Constructor } S$$
$$\text{Trait } T \quad \text{Assoc Type Constructor } D$$

$\boxed{\textsc{Terms}}$

$$\begin{aligned}
\text{Type } \tau &\longrightarrow \text{unit} \mid \alpha \mid \&\varrho\ \tau \mid \&\varrho\ \textbf{mut}\ \tau \mid \pi \mid S\langle\overline{\tau}\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \exists \alpha.\overline{p} \\
\text{Trait Instance } \mathcal{T} &\longrightarrow T\langle\overline{\tau}, \overline{\varrho}\rangle \\
\text{Projection } \pi &\longrightarrow \tau_1 . D_{\mathcal{T}}\langle\overline{\tau_2}, \overline{\varrho}\rangle \\
\text{Predicate } p &\longrightarrow \tau : \mathcal{T} \mid \tau : \varrho \mid \pi == \tau
\end{aligned}$$

$\boxed{\textsc{Declarations}}$

$$\begin{aligned}
\mathit{ctxt} &\longrightarrow \overline{\mathit{tydecl}};\ \overline{\mathit{trdecl}};\ \overline{\mathit{impl}} \\
\mathit{tydecl} &\longrightarrow \textbf{newtype}\ S\ \phi = \tau \\
\mathit{trdecl} &\longrightarrow \textbf{trait}\ T\ \phi_1\ \{\ \overline{D\ \phi_2}\ \} \\
\mathit{impl} &\longrightarrow \textbf{impl}\ \phi_1\ \mathcal{T}\ \textbf{for}\ \tau_1\ \{\ \overline{D\ \phi_2 = \tau_2}\ \} \\
\text{Parameters } \phi &\longrightarrow \forall\ \overline{\varrho}, \overline{\alpha}\ \textbf{where}\ \overline{p}
\end{aligned}$$

$\boxed{\textsc{Trait Inference Tree}}$

$$\begin{aligned}
\text{Predicate Evaluation } \mathcal{G} &\longrightarrow p \times \{\overline{C}\} \times \mathcal{R} \\
\text{Candidate Evaluation } C &\longrightarrow \mathit{impl} \times \{\overline{\mathcal{G}}\} \times \mathcal{R} \\
\text{Evaluation Result } \mathcal{R} &\longrightarrow \text{yes} \mid \text{no} \mid \text{maybe}
\end{aligned}$$

Fig. 5. A grammar for $\mathcal{L}_{\textsc{Trait}}$, the essence of Rust's trait language and inference.

3 System Design

Argus facilitates trait debugging by visualizing the entire trait inference tree in an interactive interface. The primary goal of Argus is to help developers localize the root cause of trait errors, i.e., specific failed trait obligations. Argus consists of two principal components:

(1) A Rust compiler plugin that extracts an idealized representation of trait inferences.
(2) A web-based interface for visualizing extracted trait inferences inside an IDE.

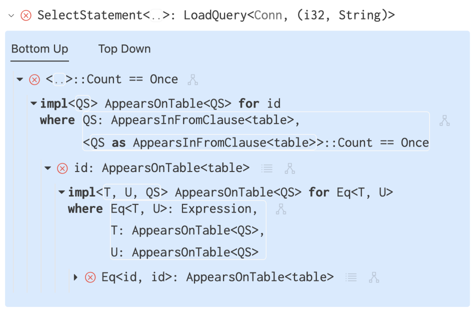
In this section, we describe the concepts most fundamental to Argus: an idealized representation of trait inferences (Section 3.1), the interface design (Section 3.2), and the heuristics used to organize information in the interface (Section 3.3). We discuss the implementation details of extracting trait inferences in Section 4.

3.1 Trait Model

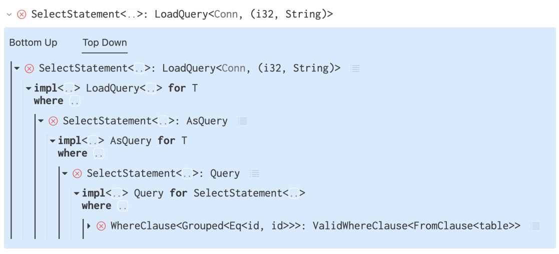
First, we need to describe the precise shape of a trait inference to understand what is being visualized in the Argus interface. Argus operates over a *trait language*, which is the subset of Rust features relevant to trait inference. This trait language is embedded within a *trait inference tree* extracted from the compiler, which can be conceptualized as a partial proof in a natural deduction system.

Figure 5 describes $\mathcal{L}_{\textsc{Trait}}$, the core syntax of Rust's trait language and trait inference trees. Rust's trait language consists of types τ, which are mostly standard with the notable addition of region-annotated references. Types are embedded in declarations, which include newtypes (*tydecl*), traits (*trdecl*), and implementation blocks (*impl*). Newtypes are relevant to the model because nominal typing permits otherwise overlapping trait implementations for the same type.

At a high level, the semantics of traits are that given a context *ctxt* and a predicate p, the compiler produces a trait inference tree $\mathcal{G}$ which describes either a successful or failed inference. An evaluated predicate $\mathcal{G}$ consists of the predicate p, a result $\mathcal{R}$, and a set of evaluated candidates C. If the predicate definitely succeeded or failed then the result is yes or no. If a predicate refers to an un-inferred type variable, then the result is maybe. A predicate evaluation succeeds if one of its


```
⌄ ⊗ SelectStatement<..>: LoadQuery<Conn, (i32, String)>
Bottom Up    Top Down
▾ ⊗ <..>::Count == Once
  ▾ impl<QS> AppearsOnTable<QS> for id
    where QS: AppearsInFromClause<table>,
          <QS as AppearsInFromClause<table>>::Count == Once
    ▾ ⊗ id: AppearsOnTable<table>
      ▾ impl<T, U, QS> AppearsOnTable<QS> for Eq<T, U>
        where Eq<T, U>: Expression,
              T: AppearsOnTable<QS>,
              U: AppearsOnTable<QS>
        ▸ ⊗ Eq<id, id>: AppearsOnTable<table>
```

(a) Expanding the inference tree in the bottom-up view.


```
⌄ ⊗ SelectStatement<..>: LoadQuery<Conn, (i32, String)>
Bottom Up    Top Down
▾ ⊗ SelectStatement<..>: LoadQuery<Conn, (i32, String)>
  ▾ impl<..> LoadQuery<..> for T
    where ..
    ▾ ⊗ SelectStatement<..>: AsQuery
      ▾ impl<..> AsQuery for T
        where ..
        ▾ ⊗ SelectStatement<..>: Query
          ▾ impl<..> Query for SelectStatement<..>
            where ..
            ▸ ⊗ WhereClause<Grouped<Eq<id, id>>>: ValidWhereClause<FromClause<table>>
```

(b) Expanding the inference tree in the top-down view.

Fig. 6. Interactions in Argus for iteratively expanding inference steps.

candidates succeeds, which in turn succeeds if all of its nested predicates succeed. Therefore a trait inference tree is an "and/or tree," of the same type found in logic program execution.

It is beyond the scope of this paper to provide a formal semantics for $\mathcal{L}_{\text{Trait}}$, e.g., a description of the trait solving process or coherence checks. The core design of Argus is largely agnostic to the internal details of the trait solver — our main focus is how to visualize the inference tree once extracted from the compiler. Instead, we will observe a few key facts about Rust's type class design which influence the kinds of trait inference trees that can emerge:

- Rust supports multi-parameter type classes. A trait can be instantiated with type parameters, and each instance is distinct from the others for purposes of coherence.
- Rust supports flexible instances and flexible contexts. Any type can be used in the "head" or "self" of an impl block and the constraints of a trait definition, so long as coherence is satisfied.
- Rust supports undecidable instances. It places no restrictions on the kinds of recursion permitted in trait bounds, as shown in Section 2.2.

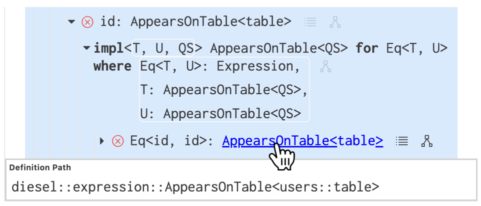
Note that we call $\mathcal{L}_{\text{Trait}}$ an *idealized* model of Rust's trait language for two reasons. First, the model omits features that are part of Rust's type system but don't meaningfully affect the design of Argus, such as constant value generics. Second, the model abstracts the complexity of the trait solver, which does not actually produce the beautiful and/or tree shown in Figure 5. We describe how to bridge that gap in Section 4.

3.2 Interface Design

The Argus interface, shown in Figure 1, takes an evaluated predicate $\mathcal{G}$ and presents an interactive visualization of $\mathcal{G}$ to the developer. The Argus interface is embedded in an IDE extension (specifically to VS Code, in our prototype) which opens a window adjacent to the developer's code when the developer's program contains a trait error. The Argus interface can also be embedded in other contexts, such as in an online textbook to pedagogically illustrate the process of trait inference in the style of recent work [10].

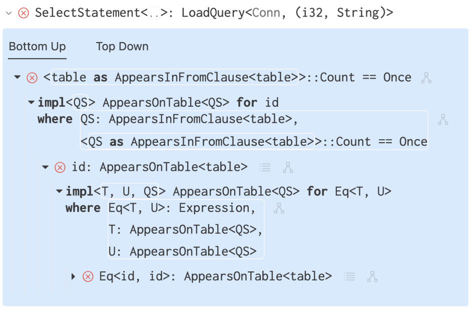
Argus is principally inspired by performance profiling tools, which also visualize a different kind of tree: a profile, i.e., a weighted call graph. Profilers can generally visualize a profile in either a top-down way (starting at the main function and descending to callees) or a bottom-up way (starting at the functions deepest in the call graph, and ascending to callers). Argus similarly exposes top-down and bottom-up views on the trait inference tree.

The details of the Argus interface are inspired by the design principles described in Section 2, which we elaborate below.



```
▾ ⊗ id: AppearsOnTable<table>
    ▾ impl<T, U, QS> AppearsOnTable<QS> for Eq<T, U>
      where Eq<T, U>: Expression,
            T: AppearsOnTable<QS>,
            U: AppearsOnTable<QS>
        ▸ ⊗ Eq<id, id>: AppearsOnTable<table>
Definition Path
diesel::expression::AppearsOnTable<users::table>
```

(a) Hovering over a type to see its fully-qualified paths in the minibuffer.



```
⌄ ⊗ SelectStatement<..>: LoadQuery<Conn, (i32, String)>
Bottom Up    Top Down
▾ ⊗ <table as AppearsInFromClause<table>>::Count == Once
  ▾ impl<QS> AppearsOnTable<QS> for id
    where QS: AppearsInFromClause<table>,
          <QS as AppearsInFromClause<table>>::Count == Once
    ▾ ⊗ id: AppearsOnTable<table>
      ▾ impl<T, U, QS> AppearsOnTable<QS> for Eq<T, U>
        where Eq<T, U>: Expression,
              T: AppearsOnTable<QS>,
              U: AppearsOnTable<QS>
          ▸ ⊗ Eq<id, id>: AppearsOnTable<table>
```

(b) Clicking an elided type to expand its definition in-place.

Fig. 7. Interactions in Argus for expanding shortened types.

3.2.1 CollapseSeq. *Instead of omitting steps of an inference sequence for brevity, allow the developer to progressively unfold the sequence.*

Unlike traditional compiler diagnostics, Argus presents an *exhaustive* view onto the trait inference tree. Every node is accessible with enough user interaction. To avoid overwhelming the developer with information, the developer iteratively unfolds levels of the tree. Argus provides two views onto the trait inference tree: bottom-up and top-down.

The bottom-up view shows the leaves of the tree first and developers can recursively expand downward towards the tree root. The developer traverses the tree from the bottom up. For example, Figure 6a shows the bottom-up view in Argus for the Diesel type error discussed in Section 2.1. Starting at the innermost failed trait bound, the developer can recursively expand its children (i.e., its parents in the inference tree) until reaching a trait bound that provides useful information about the situation, such as the `Eq<...>` type.

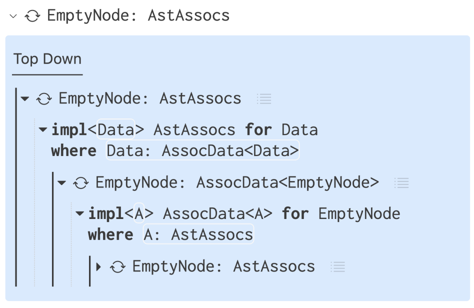
The top-down view first shows the root of the tree, i.e., the required trait bound in the program, and developers can recursively expand the children until reaching the tree leaves. For example, Figure 6b shows the top-down view in Argus for the same Diesel type error.

3.2.2 ShortTys. *Instead of heuristically shortening types, show shortened types by default, but make fully-qualified types available on-demand.*

Textual diagnostics must deal with large types via a combination of pretty-printing, heuristic shortening, and file logging. Argus instead shortens all types by default and enables developers to contextually expand them in two ways:

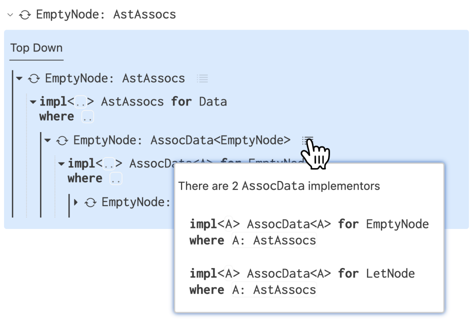
- Fully-qualified definition paths are removed, and only symbol names are printed by default. For example, the interface would print `SelectStatement` instead of `diesel::SelectStatement`. To observe the full path, the developer can hover their mouse over a symbol name, and its path appears in a mini buffer at the bottom of the page as shown in Figure 7a.
- Trait parameters, impl block quantified types, and impl block where-bounds are hidden by default, represented by an ellipsis. For example, Argus will display `SelectStatement<..>` instead of `SelectStatement<FromClause<table, ...>>`. The developer can click the ellipsis to expand out the hidden content as shown in Figure 7b.

3.2.3 CtxtLinks. *Instead of interleaving the trait inference steps with auxiliary information, enable developers to access auxiliary information on-demand through contextual links.*



```
⌄ EmptyNode: AstAssocs
Top Down
▾ EmptyNode: AstAssocs
  ▾ impl<Data> AstAssocs for Data
    where Data: AssocData<Data>
    ▾ EmptyNode: AssocData<EmptyNode>
      ▾ impl<A> AssocData<A> for EmptyNode
        where A: AstAssocs
        ▸ EmptyNode: AstAssocs
```

(a) Contextual information is hidden by default, showing only the core inference tree structure.

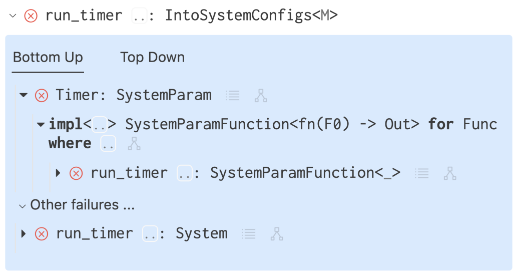


```
⌄ EmptyNode: AstAssocs
Top Down
▾ EmptyNode: AstAssocs
  ▾ impl<..> AstAssocs for Data
    where ..
    ▾ EmptyNode: AssocData<EmptyNode>
      ▾ impl<..> AssocData<A> for EmptyNode
        where ..
        ▸ EmptyNode:
```

```
There are 2 AssocData implementors

impl<A> AssocData<A> for EmptyNode
where A: AstAssocs

impl<A> AssocData<A> for LetNode
where A: AstAssocs
```

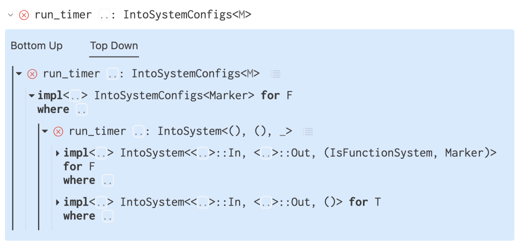
(b) The developer can query any trait bound for the full set of impl blocks for that trait, shown in a popup.

Fig. 8. Interactions in Argus for accessing contextual information about types and traits.



```
⌄ ⊗ run_timer ..: IntoSystemConfigs<M>
Bottom Up    Top Down
▾ ⊗ Timer: SystemParam
  ▾ impl<..> SystemParamFunction<fn(F0) -> Out> for Func
    where ..
    ▸ ⊗ run_timer ..: SystemParamFunction<_>
⌄ Other failures ...
▸ ⊗ run_timer ..: System
```

(a) The bottom-up view shows the deepest failed predicates, and unfolds the parents.



```
⌄ ⊗ run_timer ..: IntoSystemConfigs<M>
Bottom Up    Top Down
▾ ⊗ run_timer ..: IntoSystemConfigs<M>
  ▾ impl<..> IntoSystemConfigs<Marker> for F
    where ..
    ▾ ⊗ run_timer ..: IntoSystem<(), (), _>
      ▸ impl<..> IntoSystem<<..>::In, <..>::Out, (IsFunctionSystem, Marker)>
        for F
        where ..
      ▸ impl<..> IntoSystem<<..>::In, <..>::Out, ()> for T
        where ..
```

(b) The top-down view shows the root failed predicate, and unfolds its children.

Fig. 9. Argus visualizes two different projections of a trait inference tree: bottom-up and top-down.

The core visualization of the inference tree in Argus just shows the information strictly needed for the inference process: trait bounds and impl blocks. As shown in Figure 8a, this declutters the inference tree so that previously-obscured relationships like the overflow from Section 2.2 become simpler to visually track. All auxiliary data is instead accessible through hyperlinks and popup windows. Specifically:

- Developers can command-click any symbol to jump to its definition in the adjacent code editor.
- Developers can click a button next to each trait to access the list of impl blocks for that trait, as shown in Figure 8b.

3.2.4 *TreeData. Instead of omitting tree-shaped information in a trait inference, provide an interface that supports exploring the trait inference as a tree.*

A tree can be visualized in dozens of ways [23], with different trade-offs for the kinds of information which are easy and hard to find and understand. For example, one could imagine visualizing the inference tree in a pannable node-link diagram, which would more effectively convey the "10,000 foot view" on the inference tree. We opted specifically for a nesting-based representation because we expect that a high-level view is not particularly useful for trait debugging, since a developer most often cares about finding specific nodes in the tree. Future versions of Argus targeted at, e.g., helping Rust compiler developers design and debug the trait system itself might benefit more from a high-level view, but here we just focus on user-space debugging.

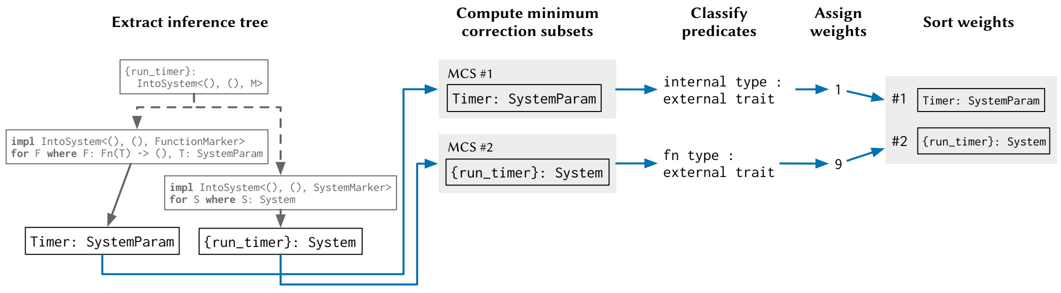

Extract inference tree
{run_timer}: IntoSystem<(), (), M>
impl IntoSystem<(), (), FunctionMarker> for F where F: Fn(T) -> (), T: SystemParam
impl IntoSystem<(), (), SystemMarker> for S where S: System
Timer: SystemParam
{run_timer}: System
Compute minimum correction subsets
MCS #1
Timer: SystemParam
MCS #2
{run_timer}: System
Classify predicates
internal type : external trait
fn type : external trait
Assign weights
1
9
Sort weights
#1 Timer: SystemParam
#2 {run_timer}: System

Fig. 10. An example of applying the inertia heuristic to the Bevy inference tree in Figure 4c. Given an extracted inference tree, Argus computes the set of smallest subsets required to satisfy the root obligation, classifies each predicate based on its structure, assigns a human-decided weight to each category, and sorts the weights.

In particular, our hypothesis is that a developer may find useful both the top-down and bottom-up views on the inference tree, depending on their specific question. A bottom-up view, as shown in Figure 9a, emphasizes most directly the possible root causes for the error. If the developer can understand these failed trait bounds without much context, e.g., by reading `Timer: SystemParam` and immediately understanding the problem, then the bottom-up view most directly facilitates fault localization. If a developer cannot understand failed trait bounds out of context, one option is to iteratively unfold the parents in the bottom-up view. Alternatively, the developer can use the top-down view to get a more “logical” view on the situation, as shown in Figure 9b. The developer reads from the visualization: we started needing to show `run_timer: IntoSystemConfigs<M>`, and that required `run_timer: IntoSystem<(), (), _>`, which could be satisfied in one of two ways, and so on.

3.3 Ranking Predicates with Inertia

The bottom-up view presents the innermost failing predicates in a particular sequence, which the developer presumably reads top-to-bottom. There is no inherent order to these predicates, because in theory each predicate could be the one that the developer intended to satisfy. In practice, we believe that some predicates are on average more likely than others to be the root cause, and that likelihood can be analyzed just from the structure of the predicate.

Our theory is that the correct fix to a failed trait error on average involves the fewest modifications to program elements, such as type definitions and trait implementations. This theory motivates our heuristic, *inertia*, used by Argus to sort failed predicates as shown in Figure 10. Inertia models the complexity of the patch required to fix a failed predicate. For Rust, we designed the inertia heuristic to reflect two common sources of complexity in fixing trait errors:

(1) *Orphan rule:* to ensure coherence of trait implementations between libraries, Rust disallowed implementing an externally-defined trait for an externally-defined type. This means a failed trait bound requiring an external type to implement an external trait requires more changes than with local types or traits (e.g., wrapping the external type in a local newtype, or changing the external type/traits).
(2) *Function traits:* higher-order functions in Rust are not generally written using function *types* as in most functional languages, but rather function *traits* due to the interaction of closures and ownership in Rust. Traits implemented for functions are written as blanket implementations like `impl<F: Fn(A) -> B> Foo for F` as opposed to hypothetically `impl Foo for fn(A) -> B`. As a result, function implementations are not rejected via unification of the head type, and they often appear as failed alternatives in trait inferences.

For example, consider the two innermost failed trait bounds in the bottom-up view on the Bevy program in Figure 9a: `{run_timer}: System` and `Timer: SystemParam`. Informally, the first bound should be higher inertia because (a) it would violate the orphan rule and (b) it involves creating a new function trait. The second bound should be lower inertia because (a) `Timer` is a local type and therefore does not violate the orphan rule, and (b) no referenced types are functions.

Formally, we compute inertia by first enumerating all minimum correction subsets (MCS). An MCS is a set of failing predicates in the trait inference tree that, if they hold true, would cause the proof to hold true. Specifically, we treat the AND/OR tree as a propositional logic formula and normalize it into disjunctive-normal form (DNF). For each conjunct in the DNF formula, we apply the inertia heuristic to compute a score for each predicate in the conjuct. The conjunct's final score is the sum of its predicate scores.

To score a predicate, we categorize it into one of eight categories of predicates. Three key categories are (1) coherent non-function trait bounds, (2) orphaned non-function trait bounds, and (3) function trait bounds. For example, `{run_timer}: System` is in category #3 and `Timer: SystemParam` is in category #1. We assigned each category a numeric rank based on the expected complexity of the fix, so e.g. category 1 is lower than 2 is lower than 3. This produces the sort order shown in Figure 9a. An exhaustive list of the categories and their ranking is provided in Appendix A.1.

4 Implementation

ARGUS is implemented as a VS Code extension that is freely available on the VS Code Marketplace and Open VSX Registry. The Rust compiler plugin is 10,393 lines of Rust code, of which 4,216 lines (40.6%) are just for serializing the Rust type system to JSON. The ARGUS interface is 8,470 lines of TypeScript code, of which 2,327 lines (27%) are just for pretty printing the Rust type system.

Beyond type serialization, the most significant implementation detail in ARGUS is how we extract the idealized AND/OR tree representation from the trait solver. One complication is that not all predicates evaluated by the trait solver represent the "final" predicates that should be presented to the developer, because trait solving and type checking are interleaving processes. It is possible that the Rust compiler provides a trait predicate with unknown type variables. Solving predicates happens in a fixpoint; ambiguous predicates remain in the trait solver queue until they are proved true or false, or until inference finishes, at which point all ambiguous predicates become failures. This reality is difficult for extensions like ARGUS because predicates re-entered into the trait solving queue are represented as new predicates. This means that ARGUS sees all snapshots of a predicate's evolution, and we use an implication heuristic to remove earlier predicates.

The second complication is the process by which trait solving and type inference guide each other. A good example is trait method calls. Consider the expression `my_value.to_string()`. Initially, there are two unknowns: the type of `my_value`, and where the method `.to_string()` comes from. Say that `my_value` has type `Vec<i32>` and two traits `ToString` and `CustomToString` provide the method `.to_string()`. The type inference engine may ask the trait solver to evaluate `Vec<i32>: ToString`, but this predicate is *speculative*. If the predicate fails, the inference engine may ask the trait solver to evaluate `Vec<i32>: CustomToString`. The issue is that all predicates, regardless of whether they are soft or hard constraints, look identical to external compiler plugins. ARGUS uses a heuristic to reverse-engineer the predicates evaluated in a program and attempts to show as few as possible. However, the version of ARGUS deployed in the user study showed potentially more failing predicates than necessary.

Finally, the grammar for $\mathcal{L}_{\text{TRAIT}}$ contains three possible predicates (trait bounds, projections, outlives-constraints), but there are actually fourteen in the Rust compiler implementation. Several of the included predicates are important details specific to Rust, but we don't want to expose them to unsuspecting developers. Argus provides a toggle setting where developers can see the full range of predicates.

Beyond a higher *quantity* of predicates, the compiler also contains *stateful* predicates, such as `NormalizesTo` τ α. This predicate is the Rust equivalent of Prolog unification, except that normalization is unidirectional. Within the compiler this predicate is used semantically like a function, where the expression τ is normalized, and the expression is written into the unconstrained type variable α. From this perspective, neither is the predicate useful nor is its subtree. Argus therefore cannot treat the entire inference tree as a tree, but rather some predicates must be treated as stateful nodes whose values can be captured only after their subtrees execute.

5 Evaluation

The central question of our evaluation is: how does Argus actually influence a Rust programmer's process of debugging complex trait errors? We explore this question in three parts:

- RQ1: How does Argus affect the overall time to localize and fix a trait error?
- RQ2: How do the features of the Argus interface individually affect a programmer's debugging process?
- RQ3: How useful is the inertia heuristic in improving the rank order of the bottom-up view?

We answer RQ1 and RQ2 by conducting a user study of $N = 25$ Rust programmers debugging a variety of trait-related errors both with and without Argus. We evaluate RQ1 quantitatively by measuring time-on-task, and RQ2 qualitatively by observing themes in participants' use of the tool. We answer RQ3 by running an experiment to quantitatively compare the relative efficacy of different predicate orderings given a ground truth specification of the correct fault.

5.1 User Study

The goal of this study was to compare Rust programmers' trait debugging strategies both with and without Argus in a variety of domains on relatively self-contained tasks.

5.1.1 Methodology.

Participants. We recruited participants from three main sources: a mailing list of Rust learners, the Rust subreddit, and the Rust Zulip. Each source provides Rust developers of different knowledge levels. The mailing list contains people with minimal Rust knowledge, the Rust subreddit contains a diverse range of experiences, and the Rust Zulip channel is mostly Rust experts and those working on the language itself. We recruited 11 participants for a trial study. Participant feedback was used to improve the materials and instructions for the final study. We recruited $N = 25$ participants for the final study. Participants had a median 11 years of programming experience (min: 2, max: 39), and a median 3 years of Rust experience (min: 1, max: 9).

The study design was reviewed by our university IRB and determined not to meet their definition of human subjects research. We nonetheless took reasonable precaution when designing and executing our user study. No personal identifiable information was collected outside of the participant's audio and screen share during the study session. Participants were compensated $20.

Materials. We created seven debugging tasks to cover a range of domains and types of trait problems. Each task consisted of a Rust crate containing one or more trait-related type errors, such as the ones shown in Section 2. The tasks contained an average of 62 lines of application code. We used two types of libraries:

- *Real libraries:* widely-used Rust libraries that make heavy use of traits, specifically: the web framework Axum [5], the game engine Bevy [2], and the SQL query builder Diesel [3]. These libraries contain an average of 25,771 lines of code.

- *Synthetic libraries:* bespoke libraries created by us for this experiment. `brew` provides an API for creating potion recipes from various plant ingredients, with invalid recipes ruled out by trait-based rules. `space` provides an API to construct intergalactic flight plans, with invalid flight plans also ruled out by traits. These APIs closely mirror the designs of Axum, Bevy, and Diesel. These libraries contain an average of 721 lines of code.

Tasks involving real libraries are maximally ecologically valid, i.e., correspond to realistic problems that Rust developers encounter. However, real libraries introduce confounds: participants may have prior experience with the library, and the quality of the documentation (e.g., prose explanation and code examples) may influence task performance. The synthetic libraries control for these factors: participants cannot have prior experience with the libraries, and the libraries only use automatically-generated documentation via Rustdoc.

For each real library, we looked at community resources and selected errors that represent common beginner mistakes. We then constructed each task by injecting a fault into a well-typed program. For example, for Bevy we used the Unofficial Bevy Cheat Book [8], which contains a section titled *Obscure Compiler Errors.* One entry is "Using a resource type directly without a `Res` or `ResMut`." We then took one of the Bevy example applications and removed the `ResMut` wrapper around the parameter of a system function. For the synthetic libraries, we injected faults that mirror the ones in the real libraries. The full set of study materials is provided in the supplementary materials.

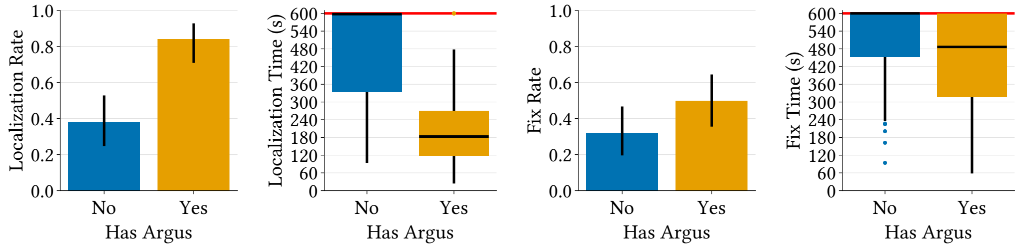
Procedure. Participants were asked to solve a series of four debugging tasks via Zoom over the span of one hour. Study sessions were conducted during one and a half months, beginning in September 2024 and running through mid-October. All sessions were recorded.

Before arriving at their session, participants read an Argus tutorial and installed the Argus VS Code extension. At the start of the session, we verbally confirmed with participants that they completed the preparation. We gave participants time to ask questions about traits, the tutorial, or Argus. We then gave a live demonstration of Argus using the first problem from the tutorial. In the demonstration, the study administrator dictated editor actions that were carried out by the participant; each action was accompanied by a reason. For example, "Please hover over the symbol `Handler` in the Argus interface — I want to see in which module it is defined."

During the study, participants were given four tasks drawn randomly from the available seven. A maximum of ten minutes was allotted per task. All debugging resources were allowed, including Google, StackOverflow, and AI chatbots / coding assistants. Participants completed four tasks total, two in each condition: with Argus, and without Argus. Task order was blocked by condition, so participants did both with-Argus tasks and then both without-Argus tasks, or vice versa based on random assignment. We asked participants to think aloud and specify (1) when they had localized the error, and (2) when they had fixed the error.

Analysis. For each task in each session we determined two values: time-to-fix and time-to-localize. Time-to-fix is relatively straightforward: we identified when the participant provided a solution that solved the type error consistent with the problem specification. This required some qualitative analysis to distinguish trivial fixes (e.g., deleting the ill-typed code) from true fixes. Note that both times are measured from the start of the task, so time-to-fix is always strictly greater than time-to-localize.

Although we asked participants to say when they localized an error, not all did, so we qualitatively coded a localization time for each task. We consider a participant to have localized a fault once they have identified the fault and started to work on a fix for the fault. For instance, in one of the Bevy tasks, the fault is that a type `Assets<Mesh>` does not implement a trait `SystemParam`. We would look



(a) Localization Rate (b) Localization Time (c) Fix Rate (d) Fix Time

Fig. 11. Distributions of localization/fix rates/times. Error bars on rates are a 95% binomial proportion confidence interval.

for indicators that the participant identified the specific issue with `Assets<Mesh>`, as opposed to the entire function or unrelated parameters. We would also look for indicators that the participant was determining how `Assets<Mesh>` could change to implement `SystemParam`.

To evaluate our ability to consistently code for time-to-localize, the second author independently coded for this variable in 20 randomly selected tasks from the dataset. The correlation between raters was $r = 0.998$ with a mean absolute deviation of $34s$. Therefore, we believe this qualitative metric can be coded with enough objectivity to be worth analyzing.

5.1.2 *Results.*

RQ1: How does Argus affect overall time to localize and fix a trait error? For both metrics, we consider its overall rate (did a participant localize/fix an error), and its overall duration (when did they localize/fix the error, capped at 10min). Localization rate and time are visualized in Figures 11a and 11b. For the localization rate, participants localized the error with Argus in 84% of cases (95% CI = $[71\%, 93\%]$), and localized without in 38% of cases (CI = $[25\%, 53\%]$), a difference of 46pp or 2.2× more cases. Using a chi-square test, this effect is statistically significant ($\chi(1, 100) = 22.24, p < 0.001$).

For the localization time, participants localized the error with Argus in a median 3m3s (CI = $[2m28s, 3m46s]$) and without was 9m58s (CI = $[7m40s, 10m]$), a difference of 6m55s or 3.3× faster. Using a Kruskal-Wallis test, this effect is statistically significant ($\chi(1, 100) = 31.39, p < 0.001$).

Fix rate and time are visualized in Figures 11c and 11d. For the fix rate, participants fixed the error with Argus in 50% of cases (CI = $[36\%, 64\%]$), and fixed without in 32% (CI = $[20\%, 47\%]$) of cases, a difference of 18pp or 1.6× more cases. Using a chi-square test, this effect is borderline statistically significant ($\chi(1, 100) = 3.35, p = 0.07$). To account for the within-subjects design, we further use a generalized linear model with condition as a fixed effect and participant ID as a random effect. Under this model, the effect is statistically significant ($p = 0.03$).

For the fix time, participants fixed the error with Argus in a median 8m7s (CI = $[6m30s, 10m]$), and fixed without in 10m (CI = $[9m52s, 10m]$), a difference of 1m53s or 1.2× faster. Using a Kruskal-Wallis test, this effect is statistically significant ($\chi(1, 100) = 5.04, p = 0.02$).

RQ2: How do the features of the Argus interface individually affect a programmer's debugging process? We answer RQ2 using qualitative observations from our user study, considering each design principle in turn.

(1) CollapseSeq: Participants iteratively unfolded the trait inference tree to varying depths, supporting the claim that there is no one-size-fits-all depth that is optimal for diagnostics.

Participants tended towards either unfolding a few steps, or unfolding the entire sequence at once. In the the former case, participants stopped early if they felt they had enough contextual information to localize an error or discard a debugging hypothesis. In the latter case, participants reported a preference for seeing all the data at once, however, they tended to spend more time debugging irrelevant information.

When first opened, the Argus interface shows the list of all bottom-up predicates together and by collapsing the inference steps the related information is viewable side by side. Not all participants started their exploration in the same place, but participants frequently collapsed inference steps after they concluded the information was no longer important. From these observations, we infer that giving participants the choice to unfold sequences is important in the exploratory phase of localization.

(2) ShortTys: For the tasks in our study, we did not observe an instance when a participant needed fully-qualified types to localize or fix a trait error, suggesting that presenting shortened types by default reduced visual noise without an adverse impact on debugging ability.

Conversely, the compiler diagnostics contain mostly fully-qualified types. Very few participants read the full error message. Developers using VS Code's diagnostic tooltips were especially likely to skip reading the diagnostic, as most of the diagnostic overflows the tooltip and is rendered offscreen.

(3) CtxtLinks: We observed that many developers preferred to look at types in in-editor source code rather than in online documentation. Participants searched source code in 73% of tasks, while documentation was opened in only 31% of tasks.

Several participants reported that if Argus was useful for nothing else, they would *still* use it for access to the source hyperlinks. This observation suggests that compiler diagnostics could benefit from a rich text representation that hyperlinks types to their source definitions.

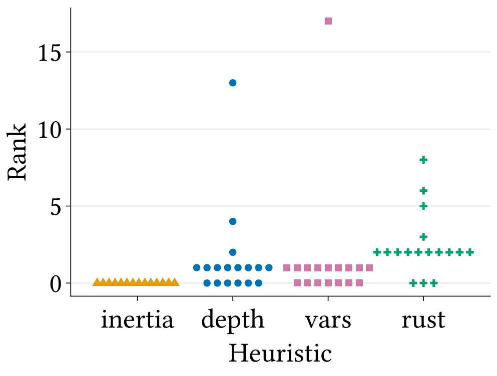
(4) TreeData: Most participants used both the bottom-up and top-down views, with a general preference for the bottom-up view (which is presented by default). Participants used the top-down view in 24% of tasks. Of those who used the top-down view, most said they preferred it for the additional context it provided. In the bottom-up view, participants generally explored the failed trait obligations from top to bottom, supporting the need for a sorting heuristic like inertia to optimize developer effort.

The specific root causes in the inference tree provide participants with information they otherwise would not have in cases such as Bevy. For tasks involving a branching inference tree, we analyzed whether each participant identified that the root cause trait (e.g., `SystemParam`) was used in the inference tree (not even that the trait was the root cause, just that it exists in the problem). Without Argus, participants only identified the trait in 29% of cases. Recall that the key trait is absent from the compiler's diagnostic, but useful to localize the error.

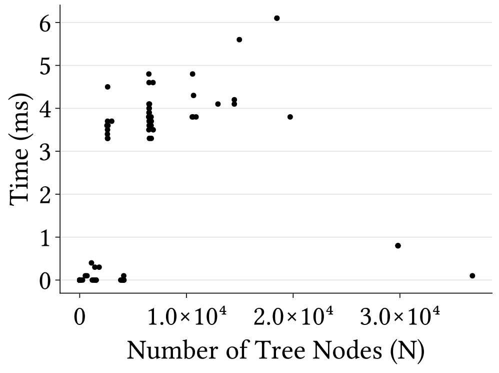
Several participants reported feeling overwhelmed by the additional information surfaced by Argus. Debugging a trait error as a tree was itself a novel idea to many participants. Because Argus exposes so much information, some participants got lost in the data and ended up debugging non-issues. It is possible that these issues can be ameliorated with more instruction and further use, but we will analyze how the interface is used by the community.

5.2 Inertia Analysis

The goal of the inertia heuristic is to increase the likelihood that the root cause of a trait failure appears near the top of the bottom-up tree view. To evaluate the efficacy of the heuristic (RQ3), we compare inertia against two categories of alternatives:



(a) The distance to the root cause for the inertia heuristic, baseline heuristics, and Rust compiler diagnostic.



(b) Time spent normalizing the trait inference trees from the test suite into DNF.

Fig. 12. Inertia experiment results.

- *Against the Rust compiler's diagnostics.* Because the compiler's diagnostics do not describe branch points, the compiler may report a failing trait bound that is higher up in the inference tree than the root cause (see Section 2.3). In this comparison, we ask: what is the minimal number of inference steps a developer would have to manually trace to reach the root failure?
- *Against simpler heuristics.* We can consider the ARGUS bottom-up view but ranked using simpler heuristics than inertia. In this comparison, we ask: if sorted by a given metric, how far down from the top would a developer have to read before reaching the root failure? We specifically consider two heuristics: depth of predicate in the inference tree, and number of uninstantiated inference variables in the predicate.

Additionally, one concern with our particular inertia heuristic is the step which converts the trait inference tree into a propositional logic formula in disjunctive-normal form (DNF). Normalizing the tree into DNF is an exponential operation, which could theoretically mean significant slowdowns for larger trait inference trees. To evaluate this performance concern, we measured the normalization time on the trait inference trees in our dataset.

5.2.1 Methodology. For both comparisons, we need a dataset of Rust programs containing trait errors where a specific failed trait bound can be blamed as the root cause of the error. We sourced the programs from Semmler's [25] database of 25 Rust programs with complex trait errors. For each program, we manually identified the trait bound in its inference tree that corresponded to the root cause of the error. We removed 8 programs for a few reasons: 2 for not having a clear program intention and error cause, 2 that are well-typed but fail to compile due to bugs in the Rust compiler, 2 for not being actual trait errors, and 2 that crash the Rust compiler. Therefore, the final suite has a total of 17 programs.

For each program, we compared against the Rust compiler by generating the trait inference tree, and counting the number of nodes between the compiler's most-specific reported error and the root cause. We compared against the alternative heuristics by computing the index of the root cause in the list sorted by each heuristic. In both cases, the optimal value is 0.

To measure performance, we profiled the time spent in DNF normalization for each program in the dataset. Performance was measured on a 2023 MacBook Pro M3 laptop with 32GB RAM.

5.2.2 Results. Figure 12a shows the distribution of distances for each approach. The median distance for each approach is 0 for inertia, 1 for predicate depth, 1 for number of inference variables, and 2 for the Rust compiler diagnostic. That is, our inertia heuristic accurately sorts the root cause to the top for every case in this particular dataset, while the other heuristics make mostly small and sometimes significant errors.

Figure 12b shows the distribution of normalization time plotted against size of inference tree. The trees in our evaluation have a median size of 2, 554 nodes (min=1, max=36, 794), and take a median $0.1ms$ (min<$0.001ms$, max=$6.1ms$) to normalize.

5.3 Threats to Validity

5.3.1 Internal Validity. Standard measures were taken to account for threats to internal validity. To account for sequencing effects, we randomized both the order of tasks the order of conditions (with vs. without Argus). Participants received training in Argus before completing any tasks, ensuring that participants did not receive different levels of exposure to trait debugging based on whether they used Argus first or last.

5.3.2 External Validity. We designed the tasks used to evaluate Argus, which could introduce bias by picking tasks favorable to our system. We combated this bias by grounding our task selection in problems identified by other people, not inventing totally new kinds of trait-related problems. Additionally, we recruited from a broad pool of Rust developers (not just, e.g., university students) so our results more likely reflect the effect of Argus in the general population of potential users.

Additionally, we designed the evaluation tasks to be comparable to our motivating examples described in Section 2, which could indicate that our evaluation does not generalize. We combat this by making Argus a general mechanism that can visualize any trait inference tree extracted from the compiler. We evaluate the system on at least the tasks it was designed for, but we cannot discount the possibility of the interface performing poorly on obscure trait errors found in the wild. We will analyze how the community uses the tool and conduct additional research should further hard-to-debug trait errors emerge.

5.3.3 Construct Validity. Localization time is a qualitatively-defined construct, which would be problematic if different people defined the point of localization differently. We checked for this consistency by measuring inter-rater reliability, finding that at least among ourselves we could consistently agree on the specific point plus/minus 30 seconds.

6 Related Work

Argus follows in a long line of systems designed to improve compiler diagnostics for type inference. Most prior work has focused on explaining failures in Hindley-Milner type inference, starting in the 1980s with the seminal work of Wand [31] on provenance-tracking for HM. Later work focused primarily on automatic fault localization by algorithmically deducing a single constraint to blame. Methods varied from SMT solving [20, 21] to Bayesian analysis [32] to machine learning [24].

The systems more closely related to Argus focus on human-centered methods of fault localization. These fall into three categories:

(1) *Improved diagnostics:* These systems retain the static text representation of compiler diagnostics, but attempt to improve the diagnostics in some manner. One approach is to include provenance information, such as the OCaml flow-based diagnostics of Bhanuka et al. [7]. We similarly believe that communicating provenance is important, but for reasons discussed in Section 2, static text is not the ideal medium to do so.

Another approach is to include domain-specific information provided by library-level annotations, such as in the Helium subset of Haskell [17], which can improve the argumentative structure of a diagnostic [6]. This is the current strategy being pursued by the Rust language developers, who recently added an `#[on_unimplemented]` attribute for custom error messages [26]. For example, the Bevy diagnostic in Figure 4b started with the phrase `` `fn(Timer) {run_timer}` does not describe a valid system configuration `` due to a library-level annotation in Bevy. This approach is largely orthogonal to Argus, which focuses on visualizing the formal structure of an inference. These approaches could also work together, e.g., if Argus used domain-specific messages to augment nodes in the inference tree. But we also believe that domain-specific annotations are not *sufficient* for diagnosing trait errors in all cases, as also shown by the Bevy example (Section 2.3).

(2) *Algorithmic debugging:* These systems present the developer a sequence of questions about predicates they expect to hold or not [28]. This approach is conceptually similar to using the top-down view in Argus and iteratively unfolding nodes, entering sub-trees if one believes a given predicate should hold. Argus improves on algorithmic debugging both by providing alternative views on the inference tree (bottom-up), but also because a graphical interface simplifies certain interactions compared to the CLI such as backtracking and exploring alternative paths.

(3) *Graphical interfaces:* These systems provide an interactive view onto the type inference process. In our review, we only identified two such published systems. First, MrSpidey [14] is a visualizer for a set-based static analysis of Scheme. MrSpidey contextually visualizes inferred abstract values, and it explains the provenance of individual constraints by overlaying arrows onto the source code. Instead, Argus opts to provide a profiler-style separate visualization of the inference tree, which we believe is more useful in practice for debugging trait inferences.

Second, the Chameleon IDE [15] is a graphical enhancement of the Chameleon system [29] for Haskell. Chameleon IDE is a single-step debugger, offering an interface for stepping through the effects of each constraint on a type inference problem. Argus does not try to present a stateful view on the inference process, but rather a projection of the final inference tree.

The vast majority of research published in this area has no human-centered evaluation of their techniques. In our review, the only systems with user studies that report task performance are Chameleon IDE [15] and OCaml flow diagnostics [7]. Notably, these studies found their tools either had extremely small effects or no significant effects on task performance, respectively. By contrast, we show in Section 5 that Argus significantly improves both the rates and duration for both localization and fixes.

In another sense, Argus is more properly placed in the dormant line of work on debuggers for logic programming. Back in the 1980s and 90s, researchers developed several tools to facilitate debugging of Prolog programs, in particular by visualizing AND/OR trees [11–13, 27]. Argus is essentially modernizing these interfaces and specializing their design to the use case of trait error debugging, while avoiding complexities of Prolog execution such as backtracking with cuts. However, if programmers continue to write ever more complex Turing-complete programs with type classes, then that line of work may provide useful ideas for visualizing type class inference as a stateful process rather than a pure natural deduction.

7 Discussion

Many prior systems have attempted to improve type inference diagnostics through increasingly sophisticated algorithms and heuristics. In this paper, we have argued that the interface matters, too. It's useful to think about inference trees as data and inference diagnostics as data visualization, especially for type classes. The results back up this argument — Argus saves developers time and energy they would otherwise waste scrounging around the documentation. In part, this is because Argus empowers developers to grapple directly with the logical structure of a trait inference, rather than falling back on heuristic reasoning based on similarity to examples. Looking forward, we hope to extend Argus to other sub-tasks of debugging (Section 7.1) and to other languages (Section 7.2).

7.1 Trait Debugging Beyond Localization

Argus primarily facilitates localization, or identifying the root cause of a trait error. However, localization is only part of debugging. As illustrated in Figure 11, many participants in our study could use Argus to successfully localize an error, but still fail to fix the error. Future iterations of Argus would ideally present more information that also facilitates fixes.

One such feature already in Argus is the ability to query for the implementers of a trait, as shown in Figure 8b. For example, if a user localizes a failed predicate like `Timer: SystemParam` in the Bevy example, then the user can inspect the implementers of `SystemParam` to potentially find alternatives like the `ResMut<...>` type. While better than nothing, this strategy is still limited because additional context is likely needed to select the appropriate implementation. Bevy provides about 30 other implementations of `SystemParam`, and it requires understanding of the library design and application design to pick among them. More generally, an open question is how to provide domain-specific feedback by making an educated guess at the user's intention from context. The `#[on_unimplemented]` feature discussed in Section 6 is one approach, but it cannot capture the full range of effective diagnostics as we have discussed.

7.2 Trait Debugging Beyond Rust

Rust's traits represent a particular configuration in the broader design space of type classes, and the Rust compiler uses a particular diagnostic approach. This raises the question: are the problems in Section 2 caused by Rust's specific design, or are they more fundamental to type classes? To what extent would the ideas in Argus be useful in other languages? We expect that for cases like the Diesel example (Section 2.1) and the AST example (Section 2.2) that the problems described are fairly general. Every type class system involves chains of inferences, and every textual diagnostic must somehow format that inference chain along with auxiliary information like source-mapping.

The Bevy example (Section 2.3) is more interesting because the core concept can be encoded differently into different type class designs. Recall that the key problem with the Bevy example in Rust is the use of an inferred marker type to distinguish otherwise-overlapping trait implementations. We examined how this problem changes when implemented in Scala, Lean, and Haskell:

- Scala's implicits seem most similar of the three to Rust's traits. Scala similarly requires a marker type to avoid conflicting `given` blocks. When confronted with a branch point, Scala's diagnostics curiously seem to pick a single branch and assume the developer intended that branch, but we could not determine Scala's algorithm for selecting the best branch.
- Lean's type classes are more expressive than Rust, as Lean has both first-class variables and permits overlapping instances (disambiguated via either names or via a numeric priority system). If we use the approach of encoding the marker type as a metavariable, then Lean returns a pithy diagnostic that again halts at the branch point: `failed to synthesize IntoSystem (Timer -> Unit) ?m.619` with no further explanation.
- Haskell's type system offers more mechanisms for manipulating type class inference than Rust, Scala, or Lean. One way to model the Bevy API in Haskell is to represent the marker type as a type family. When encoded this way, the type family eliminates the branch point in the trait inference tree, because the type of the marker is *computed* via the type family as opposed to

inferred from constraints. Subsequently, the Haskell encoding of the Bevy program generates a diagnostic that is comparably specific to Argus, i.e., that `Timer: SystemParam` is the root cause.

In sum, the design of a language's type class system will certainly affect the level of quality achievable in the language's diagnostics. More generally, modern languages increasingly use some form of search during compilation, whether that's automated theorem proving at the type-level (e.g., type classes, refinement types, tactic-based proofs) or program search/synthesis at the expression-level (e.g., supercompilation, polyhedral analysis, e-graph optimization). From the developer's perspective, all of these techniques share a common thread: it's great when they work, and hard to understand when they don't. Language designers need to understand the usability trade-offs inherent to search-based methods: they place a greater burden on the compiler to explain its work. We hope that tools like Argus help broaden our community's conception of the ways a compiler can explain itself.

Data-Availability Statement

The tutorial, study questions, raw data, and data analyses are all available in our Zenodo artifact [16]. Argus is open-source software available on GitHub [19], and the IDE extension is published and freely available on the VS Code Marketplace and Open VSX Registry.

Acknowledgments

This work was partially supported by the DARPA under Agreement No. HR00112420354 and partially supported by the NSF under Award No. CCF-2227863. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of our funders. We thank our reviewers and shepherd for their insightful and thorough feedback. Library maintainers Georg Semmler (Diesel) and Alice Ryhl (Bevy) provided early feedback and expert opinions on trait diagnostics. Members of the Rust Types Team, especially lcnr and Michael Goulet, answered numerous questions about working with the compiler and the new trait solver's interface. We are extremely grateful for the Rust community and their consistent enthusiasm to participate in user studies.

References

[1] 2022. E0275: Overflow evaluating the requirement with a generic impl. https://users.rust-lang.org/t/e0275-overflow-evaluating-the-requirement-with-a-generic-impl/73211.

[2] 2024. bevyengine/bevy: A refreshingly simple data-driven game engine built in Rust. https://github.com/bevyengine/bevy/.

[3] 2024. diesel-rs/diesel: A safe, extensible ORM and Query Builder for Rust. https://github.com/diesel-rs/diesel.

[4] 2024. rust-lang/chalk: An implementation and definition of the Rust trait system using a PROLOG-like logic solver. https://github.com/rust-lang/chalk.

[5] 2024. tokio-rs/axum: Ergonomic and modular web framework built with Tokio, Tower, and Hyper. https://github.com/tokio-rs/axum.

[6] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How should compilers explain problems to developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 633–643. doi:10.1145/3236024.3236040

[7] Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (Oct. 2023), 29 pages. doi:10.1145/3622812

[8] Ida Borisova. 2024. *Unofficial Bevy Cheat Book.* online, Chapter 4.3. https://bevy-cheatbook.github.io/pitfalls/into-system.html#obscure-rust-compiler-errors

[9] Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of type class resolution. *Proc. ACM Program. Lang.* 3, ICFP, Article 91 (July 2019), 28 pages. doi:10.1145/3341695

[10] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proc. ACM Program. Lang.* 7, OOPSLA, Article 265 (oct 2023), 29 pages. doi:10.1145/3622841 arXiv:2309.04134

[11] Alan D. Dewar and John G. Cleary. 1986. Graphical display of complex information within a Prolog debugger. *International Journal of Man-Machine Studies* 25, 5 (1986), 503–521. doi:10.1016/S0020-7373(86)80020-7

[12] Mireille Ducassé. 1998. *Abstract Views of Prolog Executions in Opium*. Research Report RR-3531. INRIA. https://inria.hal.science/inria-00073154

[13] Marc Eisenstadt and Mike Brayshaw. 1988. The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *The Journal of Logic Programming* 5, 4 (1988), 277–342.

[14] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. 1996. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 23–32. doi:10.1145/231379.231387

[15] Shuai Fu, Tim Dwyer, Peter J. Stuckey, Jackson Wain, and Jesse Linossier. 2023. ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration . In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 146–156. doi:10.1109/ICPC58990.2023.00029

[16] Gavin Gray. 2025. *gavinleroy/pldi25-artifact: v2.2.0*. doi:10.5281/zenodo.15226307

[17] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the Type Inference Process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 3–13. doi:10.1145/944705.944707

[18] Jakob Hellermann. 2024. bevycheck. https://github.com/jakobhellermann/bevycheck.

[19] Cognitive Engineering Lab. 2025. Argus: an IDE extension for debugging trait errors in Rust. https://github.com/cognitive-engineering-lab/argus.

[20] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 781–799. doi:10.1145/2983990.2983994

[21] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 525–542. doi:10.1145/2660193.2660230

[22] John C. Reynolds. 1991. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, Takayasu Ito and Albert R. Meyer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 675–700.

[23] Hans-Jorg Schulz. 2011. Treevis.net: A Tree Visualization Reference. *IEEE Computer Graphics and Applications* 31, 6 (2011), 11–15. doi:10.1109/MCG.2011.103

[24] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 60 (oct 2017), 27 pages. doi:10.1145/3138818

[25] Georg Semmler. 2022. Rust Foundation Project Grant 2022. https://github.com/weiznich/rust-foundation-community-grant.

[26] Georg Semmler. 2023. The diagnostic attribute namespace. https://github.com/rust-lang/rfcs/pull/3368.

[27] H. Senay and S. Lazzeri. 1991. Graphical representation of logic programs and their behaviour. In *Proceedings 1991 IEEE Workshop on Visual Languages*. IEEE Computer Society, Los Alamitos, CA, USA, 25–31. doi:10.1109/WVL.1991.238854

[28] Ehud Yehuda Shapiro. 1982. *Algorithmic Program Debugging*. Yale University, USA. AAI8221751.

[29] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive Type Debugging in Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 72–83. doi:10.1145/871895.871903

[30] P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. doi:10.1145/75277.75283

[31] Mitchell Wand. 1986. Finding the Source of Type Errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86)*. Association for Computing Machinery, New York, NY, USA, 38–43. doi:10.1145/512644.512648

[32] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 12–21. doi:10.1145/2737924.2738009

Received 2024-11-14; accepted 2025-03-06