ETH zürich

ETH Library

A better interface for type class diagnostics

Master Thesis

Author(s):
Gray, Gavin

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000670925

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A BETTER INTERFACE FOR TYPE CLASS DIAGNOSTICS

GAVIN GRAY

Master of Science
Department Informatik
ETH Zürich

April 2024

Gavin Gray: *A Better Interface for Type Class Diagnostics*, © April 2024

To Dad

Aunt Missy

Grandpa Louis

my Right Rib

Six months can take a lot; every day you're missed dearly

ABSTRACT

Many modern programming languages use a form of ad-hoc polymorphism as a language abstraction. Type classes, popularized by Haskell, have found space in other programming languages to join the semantics of ad-hoc and parametric polymorphism. Type class resolution is a complex black-box process implemented as a backtracking search across program types. When this search fails, compilers struggle to provide helpful diagnostics and frustrated developers have difficulty debugging complex type class errors. Textual diagnostics are fundamentally reductive and we propose an interactive interface to facilitate localization of type class errors. Our tool, Argus, is an interactive debugger for Rust traits available as a VSCode extension. For 80% of tests in a community curated test suite of difficult Rust trait errors, Argus is able to localize the root error in at most ten interface interactions.

ACKNOWLEDGMENTS

Passionate people don't let obstacles get in their way. I've somehow stumbled into a thicket of passionate researchers to whom I am immensely grateful. None of my work would be possible without Will Crichton. Given the quality of advising I've received across nine time zones, I can only imagine the energy his future PhD students will get. Will has time again saved me from the perils of JavaScript bundling as I was about to go over the edge. He's mastered the art of asking *why don't we try this*, where "this" is the exact task I wanted to avoid doing, but weeks later it turns out to be the best investment for the project.

Will and I would never have met if it weren't for Shriram Krishnamurthi. I asked Shriram if he had a well-defined Racket project I could work on and since then it's been a delightful two years bouncing around in Rust-land with Will.

I must acknowledge Peter Müller and his group for providing some of the best courses and research opportunities at ETH. I thank him for his trust as I've collaborated with researchers outside the university, red tape and bureaucracy can be hard to get through, but sometimes so worth it.

CONTENTS

1 Introduction 1
- 1.1 My Thesis . . . 1
  - 1.1.1 Overview . . . 2
- 1.2 Type Classes: a Primer . . . 2
  - 1.2.1 A Class to Stringify . . . 3

2 Failing with Class 7
- 2.1 Death by Diagnostic . . . 7
  - 2.1.1 A Good Diagnostic . . . 8
  - 2.1.2 A Poor Diagnostic . . . 10
- 2.2 Type Classes as Logic Programs . . . 13
  - 2.2.1 Type-checking a Web Server . . . 16
  - 2.2.2 Diagnosing Errors . . . 18

3 Debugging with Class 21
- 3.1 Interactive Debugging with Argus . . . 21
  - 3.1.1 A Good Diagnostic, Preserved . . . 22
  - 3.1.2 A Poor Diagnostic, Pinpointed . . . 23
- 3.2 Pulling a Rabbit Out of the Hat . . . 28
  - 3.2.1 Conjunct Analysis . . . 29
  - 3.2.2 Subgoal Analysis . . . 31
- 3.3 System Design . . . 31
  - 3.3.1 You Want What Again? . . . 33
  - 3.3.2 Moving Forward . . . 35
- 3.4 Evaluation . . . 35
  - 3.4.1 Procedure . . . 36
  - 3.4.2 Results . . . 39
  - 3.4.3 Analysis . . . 41

4 Related Work 43
- 4.1 Diagnosing Type Errors . . . 43
  - 4.1.1 Fault Localization . . . 43
  - 4.1.2 Interactive Debuggers . . . 45
  - 4.1.3 Automated Repair . . . 46

4.1.4 Domain-specific annotations. 47
4.2 Logic Programming 49
4.3 Debugging Proof Assistants 49
4.4 Final Words 50

Bibliography 53

1 INTRODUCTION

1.1 MY THESIS

Heuristic compiler diagnostics cannot always localize type class errors; interactive visualization of type class resolution is feasible and with few interactions facilitates localization.

Programmers crave abstraction. Throughout the history of computing researchers and engineers have sought to manipulate data via higher-level abstractions. One such abstraction is polymorphism, the ability for a single symbol to represent a set of types. There are several flavors of polymorphism. Parametric polymorphism, a function parameterized by a type, which includes Java generic functions or C++ templates. Subtype polymorphism, operations can act over a set of types given they're related by a notion of substitutatbility, a fancy way of saying object-oriented subtyping. Lastly, ad-hoc polymorphism, a set of functions and types that may have multiple concrete implementations, and the focus of this thesis.

IMPORTANT NOTE Accompanying the official hand-in for my thesis—this document—is an interactive website[1] to facilitate the learning outcomes for this work. PDFs no doubt contain typographic advantages as compared to other online reading media, however, the browser provides an unmatched environment for *interactivity*. The tool presented in this thesis, ARGUS, is open-source,[2] and available as a VSCode extension on both the VSCode Marketplace and Open VSX Registry.

1 https://gavinleroy.com/msc-thesis
2 https://github.com/cognitive-engineering-lab/argus

1.1.1 *Overview*

This thesis has the following structure:

- The rest of this chapter introduces type classes as a language abstraction, important terminology for discussing the language of type classes, and a basic example.
- The following, Chapter 2, introduces the structure of error diagnostics in Rust. Most important is our criteria for what distinguishes good and poor class diagnostics. The chapter finishes by demonstrating the connection between logic programming and type class systems, then proposing our solution to localize trait errors in Rust.
- After establishing the foundation for why interactive debugging benefits type class systems, we talk about the implementation of our novel tool, Argus, that provides an interactive debugging interface into Rust's trait system. We discuss the challenges and architecture of the tool as well as future plans for improvement. The chapter finishes by presenting our evaluation of the tool on a community curated test suite of hard-to-debug trait errors.
- We close with an informal look at the broader research area around interactive debugging, type error localization, and proof tree debugging.

1.2 TYPE CLASSES: A PRIMER

Coined by Strachey [23], ad-hoc polymorphism occurs when a "function is defined over several different types, acting in a different way for each type." While designing the Haskell type system, Wadler and Blott [26] introduced the mechanism of *type classes* to extend Hindley-Milner typing and unify parametric and ad-hoc polymorphism. Modern programming languages have adopted type classes to again unify ad-hoc and parametric polymorphism. Languages such as Coq, Scala, Rust, Swift, Lean, and PureScript have followed suit with varying implementations of type classes—even if branded under a different name. The remainder of this section will introduce some terminology and ordinary use cases for type classes. For the remainder of the thesis, core examples are given using Rust *traits*. In this context "trait" and "type class" are synonymous. This introduction will put Rust and Haskell side-by-side for readers familiar with Haskell.

```rust
trait ToString {
    fn to_string(&self) -> String;
}
```

```haskell
class ToString a where
  toString :: a -> String
```

Figure 1.1: Definitions of the `ToString` trait in Rust (left) and Haskell (right).

```rust
impl ToString for char {
    fn to_string(&self) -> String {
        String::from(*self)
    }
}

impl ToString for i32 {
    fn to_string(&self) -> String {
        format!("{}", self)
    }
}
```

```haskell
instance ToString Char where
  toString a = [a]

instance ToString Int where
  toString a = show a
```

Figure 1.2: Example `ToString` implementations for `char` and `i32` (a Rust integer). Trait implementations must provide all trait members, shown here is the `to_string` method.

1.2.1 *A Class to Stringify*

The simplest of our examples defines a class to turn values into a string—a common operation. To start we need to define a trait with an appropriate name and trait methods.

To minimize code, assume that the `show` *function is "built-in," and not a class member.*

Shown in Figure 1.2.1 is a trait definition, `ToString`. All future trait implementors, or class instances, will provide an implementation for each trait member—in this case, `fn to_string(&self)-> String`. A trait is hardly usefull without implementors. Figure 1.2.1 implements `ToString` for the types `char` and `i32` (a Rust integer). Both of these implementations are facts, that is, they declare that each respective type is an implementor of the `ToString` trait. (Note that the Haskell implementation `toString a = [a]` takes advantage of the fact that in Haskell `type String = [Char]`.)

The implementation for integers, `impl ToString for i32`, uses built-in formatting functions to stringify the provided integer. With some basic implementors we can already start to use the trait. Let's write a simple function, `print_ln`, that will print a value to the console.

The `print_ln` function is parameterized over a type `T`, and uses the trait method `v.to_string()` to stringify `v` then print it to standard output. However, it does so in

```
fn print_ln<T>(v: T)
where T: ToString {
  println!("{}", v.to_string());
}

printLn :: ToString t => t -> IO ()
printLn = putStrLn . toString
```

Figure 1.3: Definition of a function `print_ln` that takes type parameter `T` bound by the `ToString` trait. This function can only be instantiated with types that implement the trait bound. We call the type constraints after the `where` clause the *context*.

the *context* `T: ToString`. A context defines a set of predicates that must hold in order to invoke a function. For this function, that predicate is that the type `T: ToString` (read as "tee implements to string"). By providing this trait bound in the `where` context, Rust knows that there exists a method `to_string` on `v`.

Traits abstract over shared behavior. The `ToString` trait declares the behavior of types that can be converted into a string, trait *implementors* provide this behavior by implementing concrete member function instances. Defining every type an implementor would be a tedious process and so far we haven't seen wherein the power of traits lies. To demonstrate this, let's consider how we might use our current `ToString` implementors to stringify a vector. A naïve developer might want to provide the separate implementations shown in Figure 1.2.1.

```
impl ToString for Vec<char> {
  fn to_string(&self) -> String {
    format!("[{}]",
      self.iter()
          .map(ToString::to_string)
          .collect::<Vec<_>>()
          .join(", ")
    )
  }
}
```

```
impl ToString for Vec<i32> {
  fn to_string(&self) -> String {
    format!("[{}]",
      self.iter()
          .map(ToString::to_string)
          .collect::<Vec<_>>()
          .join(", ")
    )
  }
}
```

Figure 1.4: Bad examples of implementing `ToString` for vectors. For all vectors whose element types `T` implement `ToString`, the implementation is equivalent.

Of course this is redundant, the implementations are equivalent. Any vector whose elements implement `ToString` can share the same implementation. Traits of course allow

```rust
impl<T> ToString for Vec<T>
  where T: ToString, {
  fn to_string(&self) -> String {
    format!("[{}]",
      self.iter()
          .map(ToString::to_string)
          .collect::<Vec<_>>()
          .join(", ")
    )
  }
}
```

```haskell
instance ToString a => ToString [a]
    where
  toString a = "[" ++ listWSep ++ "]"
    where
      listOfStrs = map toString a
      listWSep = intercalate ", "
          listOfStrs
```

Figure 1.5: Improving upon the bad example in Figure 1.2.1 the trait implementation can be parameterized to accommodate both `Vec<char>` and `Vec<i32>`. This parameterized implementation uses a context to place the trait bound `T: ToString` requiring that `T` provide an implementation of the `ToString` trait.

for this abstraction, implementations can be *parametric*, thus chaining implementation blocks together. The previous two declarations would canonically be implemented as in Figure 1.2.1.

Similar to the definition of `print_ln` earlier, the implementations in Figure 1.2.1 are parametric over `T` and rely on a context. Writing this declaration in English we might say "a vector implements `ToString` if its elements implement `ToString`."

The trait machinery of Rust will "figure out" which implementations to call when `print_ln` is invoked with vectors of characters or integers. The figuring out process in Rust is called *trait resolution*. A complex process and the topic of this work.

```rust
print_ln(vec!['a', 'b', 'c']) // "[a, b, c]"

print_ln(vec![1, 2, 3]) // "[1, 2, 3]"
```

Figure 1.6: Example invocations of the parametric function `print_ln`. The Rust compiler resolves implementations for `char: ToString` and `i32: ToString` allowing a vector of these elements to also implement `ToString`.

2 FAILING WITH CLASS

Type classes as an abstraction feature is prevalent in many modern programming languages. As discussed in Section 1.2, class instances are not always simple declarations, but they can be polymorphic, thus chaining together. These polymorphic instances breathe life to the power of type classes, but they also create a tower of abstraction that, when fails, compilers cannot always reduce to a helpful error diagnostic.

For the scope of this work, a trait error has occurred if the language cannot *prove* that a type `U` implements trait `T`. This definition does not state that `U` *isn't* an implementor of `T`, rather, that its existence cannot be proven.

We outline our criteria for a good error diagnostic. We call a diagnostic good if it localizes the root cause of the failure and provides full provenance for all required trait bounds. Poor diagnostics fail to provide this information. We will show how branching in the Rust trait solver impedes its ability to localize the root cause of an error. Language design decisions impact which patterns are preferred by library maintainers. This section highlights some common patterns in Rust and how the interplay between language features and trait resolution lead to poor diagnostics.

2.1 DEATH BY DIAGNOSTIC

Many programmers laud the Rust language for providing "good" error diagnostics. All languages should strive to provide insightful diagnostics but in this section we will see how trait resolution can easily leave developers with little debugging information.

Traits in Rust are interesting for several reasons. The language is gaining popularity in "the mainstream" and developers come to Rust from dynamically-typed languages, such as Python and JavaScript, or from traditional low-level systems languages like C. Rust may be the first strong statically-typed language they learn and their first encounter with traits. The trait system provides a flexible semantic compared to Haskell '98, but with added flexibility comes undecidability. [15] Rust developers struggle to debug trait errors especially in the face of complex trait systems that may overflow. For example, Diesel is a object relational mapper and query builder for Rust that uses traits extensively for static safety. Currently, five of the twenty discussion pages are

```
fn main() {
  print_ln(vec![1, 2, 3]); // "[1, 2, 3]"

  print_ln(vec![1.618, 3.14]); // Whoops!
}
```

Figure 2.1: The trait `ToString` is not implemented for `f32`, therefore the second function call results in a trait error.

questions related to trait errors. [29] Traits are a key abstraction in Rust and many other language features rely on them, closures, async, and thread safety to name a few. On a quest for strong static guarantees, many trait-heavy crates are emerging in the Rust ecosystem, forcing developers to confront complex systems of traits to do something as simple as matrix multiplication. There is sufficient evidence to suggest that trait errors can be hard to debug—by newcomers and experts alike.

2.1.1 *A Good Diagnostic*

Our interest now turns towards diagnostic messages. Specifically what is meant by the terms "good" and "poor," and what distinguishes the two. Exploring first with the current example, shown in Figure 2.1.1 `print_ln` is called with a vector of *floating-point* numbers (or `f32` in Rust). This results in a type error because `f32` does not implement the `ToString` trait.

Rust diagnostics follow a particular structure. There's a principle message , followed by a series of notes on the origins of this principle. Often included are suggestions, or "help" messages for what might fix the problem . In Figure 2.1.1 the principle message states that the trait bound `f32: ToString` is not satisfied. This unsatisfied bound is the *root cause* of the type error. Rust then suggests other types that do implement `ToString` , in this case, `i32` , if perhaps that was your intention.

The provenance of trait bounds is important. Why does `f32` need to implement `ToString` ? Well, the compiler wants to use the implementation rule for `Vec<T>` which requires the bound on `f32` as a subcondition.

The final piece of provenance is where the initial `ToString` bound was introduced. The condition on the instantiated generic type, `T` , when calling `print_ln` .

We label this as a *good* diagnostic message. Good diagnostics help developers accomplish a task, usually by revealing a hole in their mental model or answering a

```
error[E0277]: the trait bound `f32: ToString` is not satisfied
   |
40 |     print_ln(vec![1.618f32, 3.14])
   |     -------- ^^^^^^^^^^^^^^^^^^^^ the trait `ToString` is not implemented for `
       f32`, which is required by `Vec<f32>: ToString`
   |     |
   |     required by a bound introduced by this call
   |
   = help: the trait `ToString` is implemented for `i32`
 note: required for `Vec<f32>` to implement `ToString`
   |
17 | impl<T> ToString for Vec<T>
   |         ^^^^^^^^     ^^^^^^
18 | where
19 |     T: ToString,
   |        -------- unsatisfied trait bound introduced here
 note: required by a bound in `print_ln`
   |
32 | fn print_ln<T>(v: T)
   |    -------- required by a bound in this function
33 | where
34 |     T: ToString
   |        ^^^^^^^^ required by this bound in `print_ln`
```

Figure 2.2: Calling `print_ln` with `Vec<f32>` results in a trait error. `f32` does not implement the trait `ToString`, therefore, the implementation rule for vectors cannot be used. The resulting error diagnostic properly reports the root cause, and provides the full provenance of trait bounds. We call this a *good* diagnostic.

question. Given a developer's expectation for type U to implement trait T, potentially via transitive implementors $A, B, C, \ldots$, a diagnostic should explain where it failed in that chain. To accomplish this task there are two essential components for a diagnostic to provide: the root cause, and its provenance.

SPECIFIC ROOT CAUSE The principle message reported: "trait bound `f32: ToString` not satisfied" is as specific as it gets. The compiler could have reported "the bound `Vec<f32>: ToString` was not satisfied," but this isn't as specific. The more specific the reported root cause, the faster developers can address the issue. There isn't however a precise definition of root cause. As previously mentioned, diagnostics should help uncover holes in a developer's mental model. Therefore, the root cause is a function of the developer's mental model of the program and not solely the program per se. In general we cannot expect a computer-generated diagnostic to always point to the root cause because it lacks access to the developer's mental model.

FULL PROVENANCE The full provenance should answer the question "Why is this bound required?" Developers should not be left unknowing of where a trait bound came from. Good diagnostics provide this provenance. `f32: ToString` was required because it's a condition on the implementation block for `Vec<T>: ToString`. Furthermore, `Vec<T>` was required by the condition on the call to `print_ln`. In the diagnostic Rust has laid out this path for developers to follow.

2.1.2 *A Poor Diagnostic*

For decades programmers have laughed—and cried—at pages of unreadable diagnostics spewed by the C++ and Java compilers. Compilers have the Herculean task of packaging complex failures into digestable and helpful diagnostics—a task that increases in complexity as type systems become more sophisticated. Modern type-checking involves proof trees and SMT queries; a type error can often be a single bit response "No," leaving a frustrated programmer to trial-and-error debugging. In this section we will see how a system of traits can leave Rust programmers in a similar situation.

Axum is the most popular web framework in Rust that touts its focus on "ergonomics." Using Axum is easy. To demonstrate, our second running example will use Axum to create a simple web. Servers provide a set of routes, and each route has a

```
async fn login(name: String, pwd: Bytes) -> String
    {
    if is_valid_user(&name, &pwd).await {
        format!("Welcome, {name}")
    } else {
        "Invalid credentials".to_string()
    }
}
```

```
use axum::{
    body::Bytes,
    routing::post,
    Router
};
use tokio::net::TcpListener;

async fn is_valid_user(
    name: &str, pwd: &Bytes
) -> bool { true }
```

```
#[tokio::main]
async fn main() {
    let app = Router::new()
      .route("/login", post(login));
    let listener = TcpListener::bind("0.0.0.0:3000")
        .await.unwrap();
    axum::serve(listener, app)
      .await.unwrap();
}
```

Figure 2.3: Simple web server example using the Axum framework. The `login` function is intended for use as a post handler, except it does not satisfy the constraint of the trait system. Handlers are asynchronous function with 0–16 parameters, each of which must be an *extractor*. The bound `String: FromRequestParts` is not satisfied and therefore `login` cannot be used as a handler. The Rust compiler reports the general failed bound, `login: Handler`.

message handler. Creating a handler is as easy as: defining an asynchronous function that has 0–16 "extractors" as parameters and returns a type convertable into a web response—simple. Shown in Figure 2.1.2 is an example server that provides a route for users to login to an artificial application.

There is a some boilerplate required to make a server, but the important piece is the asynchronous function `login`. This is the handler for the `/login` route. This handler takes two extractors. The intention is that `String` will extract out a string from the incoming message, to be the user's name. The second extractor, `Bytes`, consumes the remainder of the message that will be the user's password. (Encrypted of course.) The response type, `String`, is returned by the function body —indicating success or failure. This code does not compile. When used as a handler parameter `String` must come *last*. To understand why, we first need to dig into some of the Axum trait specifics.

```
error[E0277]: the trait bound `fn(String, axum::body::Bytes) -> impl Future<Output =
    String> {login}: Handler<_, _>` is not satisfied
    |
22  |     let app = Router::new().route("/login", post(login));
    |                                             ---- ^^^^^
    |     the trait `Handler<_, _>` is not implemented for fn item
    |     `fn(String, axum::body::Bytes) -> impl Future<Output = String> {login}`
    |                                              |
    |     required by a bound introduced by this call
    |
    = help: the following other types implement trait `Handler<T, S>`:
              <Layered<L, H, T, S> as Handler<T, S>>
              <MethodRouter<S> as Handler<(), S>>
```

Figure 2.4: A poor error diagnostic resulting from an improper use of the `login` function that doesn't implement the trait `Handler`. Parameters of type `String` must come last in handler signatures, because they consume the entire message body. `login` tries to use both strings and bytes, both of which consume the message body. The root cause is the failed bound `String: FromRequestParts`, but the high-level bound `login: Handler` is reported.

This example demonstrates all the necessary pieces of a handler. Extractors are doing some heavy lifting, the types encode how an incoming message is parsed into separated parts, and their explanation, till now, was surface level. One tricky "gotcha" with extractors is that the first $n - 1$ parameter types must implement the trait `FromRequestParts`, and the nth must implement the trait `FromRequest`. This distinction helps Axum peel off *parts* of a message, then consume the rest with the last parameter. Types implementing `FromRequestParts` do the peeling. Types implementing `FromRequest` do the consuming.

As stated, strings must come last. A `String` does not implement `FromRequestParts`. It does however implement `FromRequest` and can be used to consume the remainder of a message. With this knowledge, in an error diagnostic we should hope the principle message is: " `String` does not implement `FromRequestParts`." Instead, we get the paragraph shown in Figure 2.1.2.

The principle message provides the unhelpful, in lay man's terms, " `login` is not a `Handler`." The important `FromRequestParts` trait makes no appearance.

Fortunately, some provenance is provided . This short provenance describes why `login` must implement `Handler`, but again, the information stops there and no deeper cause is mentioned.

Programmers have a mental model. The diagnostic has said "your function is not a handler," but as the programmer I *intended for it to be*. A good diagnostic should provide information on why my mental model was violated. I want `login` to be a handler. It isn't, but how can I figure out where I went wrong? The first criterion for a good diagnostic message is violated. The compiler has not provided the root cause, anywhere. There is some provenance, but not *full provenance* to the root cause. The Axum diagnostic has failed the criteria and is therefore a poor diagnostic.

There is an explanation for why Rust provided a good diagnostic in the first example and not in the second. This explanation lies within *trait resolution*, the process by which Rust decides which trait implementations will be called at runtime.

2.2 TYPE CLASSES AS LOGIC PROGRAMS

Type class systems are logic languages. A specific set of trait definitions and implementors form a logic program. Trait resolution then evaluates the constructed program and is implemented as a backtracking search. We reduce trait resolution to logic programming to emphasize problem structure. This problem is not inherent to Rust per say, but a broader complexity coming from this evaluation style. Logic languages provide a language-agnostic way to discuss type class systems. By abstracting away Rust-specific details we highlight that this problem is inherent to any type class implementation reducible to logic program evaluation. To explore the difference between the good and poor diagnostics of the previous section we will turn each set of traits and implementations into a logic program. We then interpret a query over this program to simulate trait resolution.

Our toy example motivating the need for type class abstraction, the `ToString` trait, produced a good diagnostic. Let us step through the structure of this program one more time, but instead we will focus on the declarative semantics of the trait language.

A trait definition establishes a unit of shared behavior. We can ask questions like "Does this type implement that trait?" And thus is akin to a single-arity Prolog predicate. Using Prolog conventions we have the `to_string/1` predicate that declares its parameter an implementor of the `ToString` trait.

We can provide a trait implementation for ground types. Shown in Figure 2.2 are the implementations for `i32` and `char`. These implementations declare *facts*. A fact that these types satisfy the `to_string` predicate.

Non-parameterized types are converted to atoms in the logic model.

```
trait ToString { /*...*/ }

impl ToString for i32 { /*...*/ }

impl ToString for char { /*...*/ }

impl<T> ToString for Vec<T>
where
  T: ToString, { /*...*/ }
```

```
to_string(i32).

to_string(char).

to_string(vec(T)) :-
  to_string(T).
```

Figure 2.5: A translation of the `ToString` trait and its implementors into a logic program. Traits become predicates, non-parameterized implementations become facts, and parameterized implementations become rules. Trait solving can be thought of as querying the logic database, for example `?- to_string(vec(f32))` is equivalent to asking "does `Vec<f32>` implement `ToString`?"

With polymorphism we can chain trait implementations, such as the type-parameterized implementation for `Vec<T>`. These implementations form a *rule*. In order for the rule to be proven, several additional body predicates need to be satisfied. These predicates are the clauses that come after Rust's `where` keyword, or after Prolog's `:-` in the rule body.

Calling our `print_ln` function with a vector of floating-point numbers created a trait bound, `Vec<f32>: ToString`. It is then the compiler's job to resolve the `ToString` implementation for the instantiated type `T`. In logic programming we can phrase this bound as a *query*, `?- to_string(vec(f32))`, interpreting this query is equivalent to resolving the trait implementation.

The funny `?-` indicates that the clause is a question, not a statement.

We take the definition of logic program computation from Sterling and Shapiro [22]: computation progresses via *goal reduction*. At each stage there is a conjunction of goals to be proved, this is called the resolvent. The interpreter will choose a goal from the resolvent and a clause from the program environment such that the clause head unifies with the goal. (A clause is either a fact or a rule.) The computation continues by replacing the resolvent with the clause's body. When picking a goal and clause, we constrain the interpreter to choose left-to-right and top-to-bottom, respectively. After successfully proving a goal in the conjunction the interpreter continues to the next. After failing to prove a goal the interpreter *backtracks* to the last point of choice, where a clause was picked from the environment, and chooses the next clause if one is available. Tracing the execution of the described interpreter constructs a *proof tree*. Each goal has a set of candidate clauses from which the interpreter can choose to perform goal

```
to_string(i32);
to_string(char);

to_string(vec(T)) :-
   to_string(T);

?- to_string(vec(f32)).
```

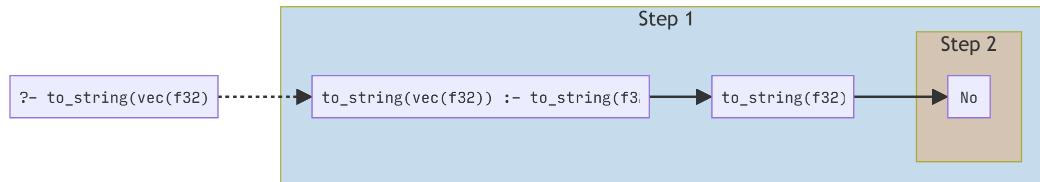


Figure 2.6: Execution trace of the logic interpreter for the `ToString` example. Step 1: The goal unifies with the rule head from the program environment. The goal is then reduced to the conditions in the rule body, here the singular `to_string(f32)`. Step 2 The goal does not unify with any program clause and execution fails.

reduction. Reducing a goal with a rule clause introduces a conjunction of subgoals that must be proven. If a goal fails, the interpreter can *backtrack* and retry by reducing the goal with a different clause. This was quite a bit of technical information, but the visualization clarifies the proof tree structure.

Tracing the execution of the simple program using `ToString`, we get the proof tree shown in Figure 2.6, goals connected by a dashed line represent an Or relationship. At least one child in an Or relationship must hold for the parent to be proven. Goals connected with a solid line form an And relationship. All child goals must be proven for their parent to be proven. This type of proof tree is commonly referred to as an *And–Or tree*. Each level in the tree is either a conjunction: all child goals must be proven, or a disjunction: at least one child goal must be proven.

The execution of this program was as follows. The start of the search process is the root goal. This is where the Rust trait solver asks “Does `Vec<f32>` implement `ToString`?” The interpreter unifies the goal with the rule head from the program. Unifying `to_string(vec(T))= to_string(vec(f32))` results in the constraint `T = f32`. The current goal is reduced to the body of the rule, `to_string(f32)`. In this case there is

only one goal of the rule body, if there were many, they would be conjunctive. These conjunctive nodes are siblings in the proof tree and they form an And relationship with their parent. This means that for the parent goal to be proven, all children must be proven. The now current goal, `to_string(f32)`, fails to unify with any clause in the program. Therefore this goal cannot be proven. There are no other rules the interpreter can backtrack to. Therefore, the root goal, `to_string(vec(f32))` cannot be proven. It's then said that the trait bound `Vec<f32>: ToString` *is not satisfied*.

Now that terminology is out of the way and we've seen a simple example, it's time to start looking at the proof tree for the Axum trait error. The resulting proof tree for the simple `ToString` trait was more of a *proof stick*, no backtracking occurred. Let us direct our attention to the running example with Axum, introduced in Section 2.1.2, where we will see some branching.

2.2.1 *Type-checking a Web Server*

We start with the root goal, asking whether the type `login` implements `Handler`. In Rust, `login` has type[1]

```
fn(String, Bytes) -> impl Future<Output = String>
```

Our interpreter chooses clauses top-to-bottom. Step 1 The first rule `handler(fn_once([T1], Fut))` is tried but automatically fails. We've said that the rule head needs to unify with the goal and in this case the arity of the goal and rule head functions differ—two vs one. Continuing its search the interpreter *backtracks* and tries to reduce the goal via a different clause, the second implementation rule for `handler`. Step 2 The second clause head does unify with the current goal. Take note that these two rules form an Or relationship with the root goal. The first failed but the root goal may still be proven via the second clause. After the clause head unified, the goal is replaced by the clause body. The subgoals introduced in the reduction are a conjunction, forming an And relationship with the parent rule.

Step 3 Working left-to-right, the goal `future(promise(string), Res)` needs to be proven as part of the conjunction. The interpreter unifies this with a program *fact* and the subgoal is said to be proven. The effect of this unification is the learned constraint, `Res = string`. Continuing in the interpretation we now have to prove the subgoal

1 The logic version of the Axum API uses the atom `promise` to represent something that implements the future trait. This is a bit over-simplified from the Rust Async model, but an unimportant detail for this example.

```
handler(fn_once([T1], Fut)) :-
    future(Fut, Res),
    into_response(Res),
    from_request(T1).

handler(fn_once([T1, T2], Fut)) :-
    future(Fut, Res),
    into_response(Res),
    from_request_parts(T1),
    from_request(T2).
```

```
fn_once([], _).
fn_once([_|_], _).

from_request_parts(parts(_)).
from_request(bytes).
from_request(string).

future(promise(O), O).
into_response(string).
```

```
?- handler(fn_once([string, bytes]), promise(string)).
```

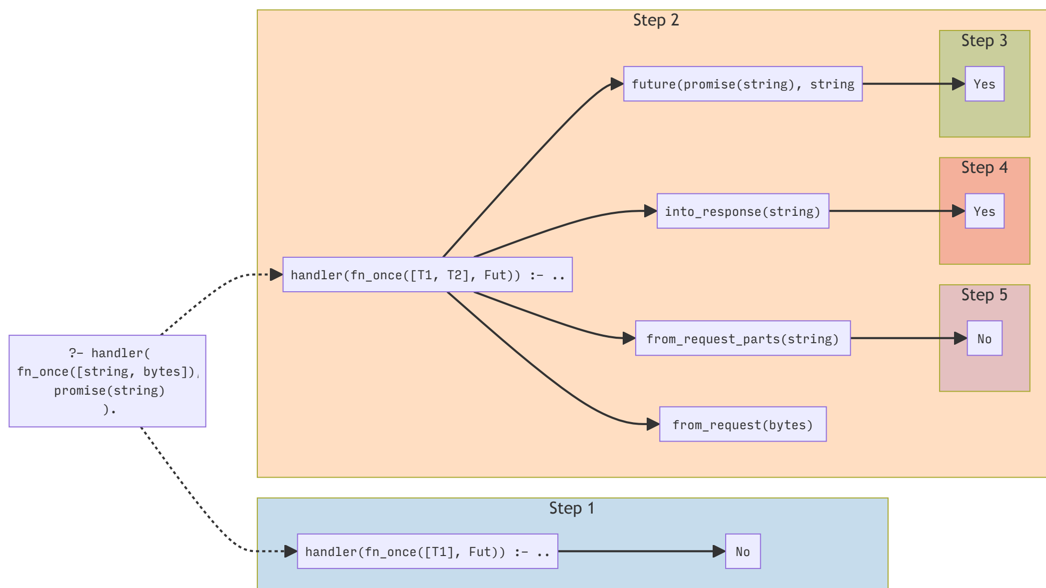


Figure 2.7: Execution trace for interpreting the query created by the trait bound `login: Handler`. Step 1 The solver attempts to unify the goal head with the rule body but the mismatch in function arity causes the unification to fail immediately. Step 2 The solver *backtracks* and unifies the goals with the second rule head, this time succeeding. The goal is then reduced to the subgoals of the rule body. Step 3 The response type unifies successfully with the function return value, and the constraint `Res = string` is added. Step 4 The resolvant subgoal `into_response(string)` unifies with a program fact. Step 5 The next subgoal, `from_request_parts(string)` fails to unify with any program clause, therefore, this subtree fails. The interpreter has no further rules it can backtrack to and therefore the root goal fails.

`into_response(string)`. Step 4 One again a fact from the program unifies with the goal and it is said to be proven. Step 5 The first parameter type of the function `login`, `String` must implement `FromRequestParts`. The goal, `from_request_parts(string)`, *does not* unify with any program clause. Therefore, the goal *fails*. At this point, the interpreter has exhausted the disjunction of clause rules and can no longer backtrack. The root goal can therefore also not be proven and execution halts.

`IntoResponse` is the set of types the Axum API knows how to convert into an HTTP response.

2.2.2 *Diagnosing Errors*

Each of the running example programs was translated into an equivalent logic program, and the queries were traced to build proof trees. Both trees in Figure 2.6 and Figure 2.7 failed, and the compiler must turn this proof tree into a diagnostic message. We can partially formalize this final step in the compiler as a function, `diagnose`, that takes a tree as input and returns the problematic tree node.

```
diagnose :: ProofTree -> Node
```

The `diagnose` function is *reductive*. It takes the large data of a proof tree and chooses a small piece to show developers. There is a tension in forming error diagnosis as a reduction. Diagnostics want to be as helpful as possible, reflect on our criteria for a good diagnostic: it must determine the specific root cause and includes *full provenance* of the failure. Shown in Figure 2.8 are the two failing proof trees for the good and poor diagnostics.

The `ToString` examples produces a simple tree. There only exists a single failing node! In this case the reduction is straightforward, by reducing the tree to its single failing node, `to_string(f32)`, no information has been lost. The specific root cause is the failing node and the provenance is generated by traversing the tree upwards, from failing node to root. The complex tree produced by the Axum `Handler` example is not as straightforward. Here the reduction must choose between two failing nodes, however, choosing either of the nodes forgets the information of the other failed branch. Compiler diagnostics are conservative, so rather than reduce the tree to a failed node the tree is reduced to the *least common ancestor* of the failed nodes.

One's initial intuition may be to simply improve reduction algorithm. This is equivalent to "making diagnostics better." Indeed there are circumstances where type information may be able to eliminate certain failed branches, but the problem is still that of reduction.

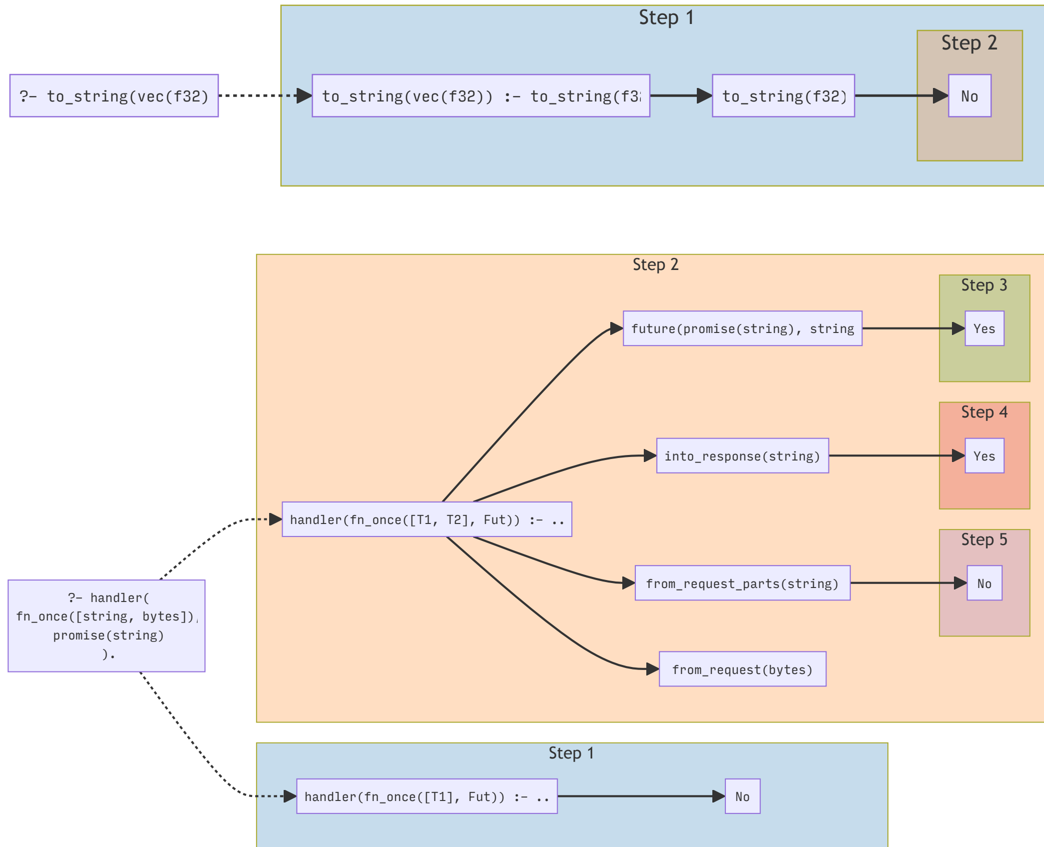


Figure 2.8: Proof trees obtained from tracing the logic interpreter (or equivalently the trait solver). for the and examples. Top: simple trace for `ToString` with no branching, that produced a good error message. Bottom: trace of the complex `Handler` example. This trace does contain branching and the error diagnostic did not mention any data *below* the branch point.

This thesis proposes to change the fundamental problem of error diagnosis. We proposes an alternative function, `ui`, to the reductive `diagnose`:

The type constructor `Visualize` is doing a lot of heavy lifting in this signature.

```
ui :: ProofTree -> Visualize ProofTree
```

No longer is error diagnosis a reduction problem. By changing the output type from `Node` to `Visualize ProofTree`, the problem is now an *interface problem*. We want to facilitate information extraction from the proof tree and allow developers to discover the root cause of an error, while preserving provenance. The benefit is that no information is lost; the root cause of an error is always available. The drawback, again, is that no information is lost. Proof trees are large and complex, and the designed interface should facilitate localizing the root cause with as few interactions as possible.

3 DEBUGGING WITH CLASS

Interactive visualization of type class resolution is feasible and with few interactions facilitates localization. This chapter presents Argus; a tool to facilitate interactive exploration of trait resolution in Rust. Argus does not provide error diagnostics in the traditional sense, but features that satisfy our criteria for a good diagnostic: access to the root cause of a trait error, and its full provenance. The interactive interface facilitates exploring an error in the way a static compiler diagnostic cannot.

In Section 2.2 we established that sets of traits and implementors are logic programs. Trait bounds, or obligations, are logic queries that evaluate the program. Debugging logic programs is difficult, and by equivalence, so is debugging arbitrary trait errors. Literature from algorithmic debugging and interactive fault localization provides a basis on which we can build an interactive tool to debug trait errors.

Reinterpreting diagnostics as an interface problem introduces the challenges and drawbacks of visualizing large data. Not only does Argus provide an exploratory interface of proof trees, but also a set of simple heuristics to suggest developers where to start in the debugging process. The Rust community has created a set of confusing trait errors, and for 80% of tests developers using Argus will encounter the root cause within 20 cognitive steps. This chapter outlines our design and development of Argus and how a generalized tool of this nature could benefit all languages with class—type class.

3.1 INTERACTIVE DEBUGGING WITH ARGUS

Argus is a tool for interactive exploration of Rust's trait solving process, made possible by exposing the internal proof trees constructed by the Rust trait solver. The Argus interface creates a lens into trait resolution and does not reduce away important information like compiler diagnostics. This section introduces the Argus interface with the two running examples, and shows how Argus can provide more specific error information for our Axum web server.

An interactive version of this thesis is available online and recommended over the PDF—pictures aren't quite as fun as the real thing.

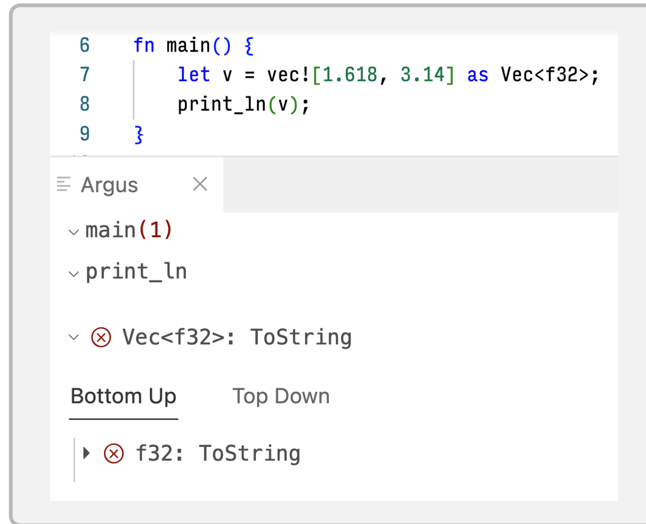


Figure 3.1: The Argus bottom-up view for the simple `print_ln` error seen in Figure 2.1.1. The root cause, the failed bound `f32: ToString` is the only failed subgoal in this simple example.

3.1.1 *A Good Diagnostic, Preserved*

Compiler diagnostics are reductive, explained in Section 2.2.2. If a reductive diagnostic can satisfy our criteria for a good error message, an interactive interface should *do no worse*. Simple cases should remain simple, at the very least Argus should never perform worse than a compiler diagnostic message.

Shown in Figure 3.1 is the default Argus interface. Failed obligations are grouped by expression of origin (where the obligation was introduced), expressions are grouped by functions, and functions are grouped by file. A key idea in Argus it that there is not a single correct way to view the proof tree. The best view depends on the programming task. Similar to a performance profiler, Argus presents both a *bottom-up* view (starting at the failed leaves) and a *top-down* view (starting at the root goal). By default Argus starts in the bottom-up view. Recall that we claim: interactive visualization facilitates error localization with few interactions. By presenting the bottom-up view first, developers are immediately confronted with the potential root causes.

Advanced Rust users may have other top-level items such as constant expressions.

For this simple trait error, the bottom-up view shows us the same information as provided by the compiler diagnostic: an unsatisfied trait bound `f32: ToString`. That is

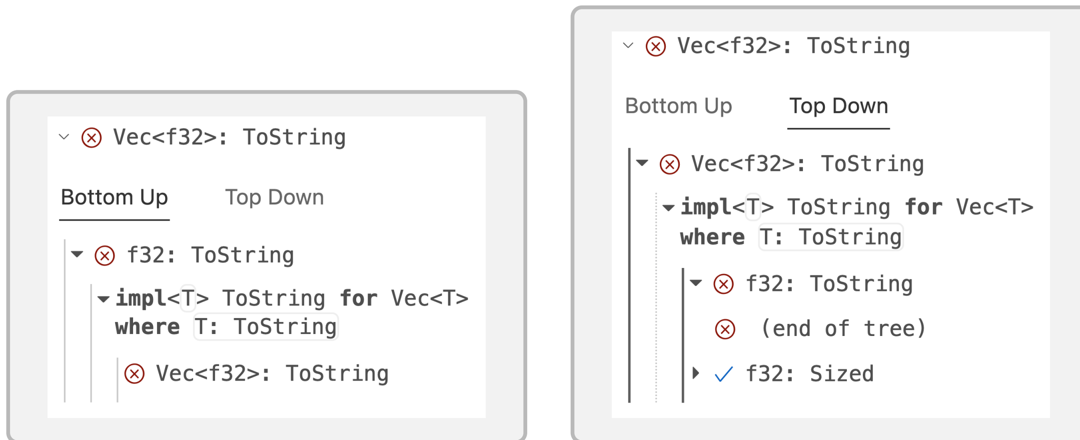


Figure 3.2: The expanded versions of the bottom-up (left) and top-down (right) views for the root cause `f32: ToString`. The trait solver did not branching in this example so the views are mirrors of each other. The top-down view encodes the relationship (And/Or) of the child to its parent. A dashed line signifies an Or relationship and a solid line signifies an And relationship.

our root cause. Expanding the bottom-up view shows the provenance of each bound. The top-down view of this example is uninteresting because the trait solver did no branching, and the proof tree is really a "proof stick." Therefore, it is simply the mirror of the bottom-up view as shown in Figure 3.2.

The top-down view starts with the failed root goal, and interactively allows for exploration down towards failures. Encoded in the left border is the relationship between parent and child (And/Or). A dashed border represents the Or relationship, meaning one child needs to hold. This is used when reducing a goal via a trait implementation rule. The solid border represents the And relationship, meaning all children need to hold to satisfy the parent. This conjunctive relationship is most commonly introduced by goal subconditions introduced by the `where` clause of a trait implementation block.

3.1.2 *A Poor Diagnostic, Pinpointed*

In Chapter 2 we explored how the requirements for the Axum `Handler` trait created a poor diagnostic. Figure 2.7 traced the trait solver execution; due to branching and backtracking in the search the compiler reports a higher failed bound than the actual

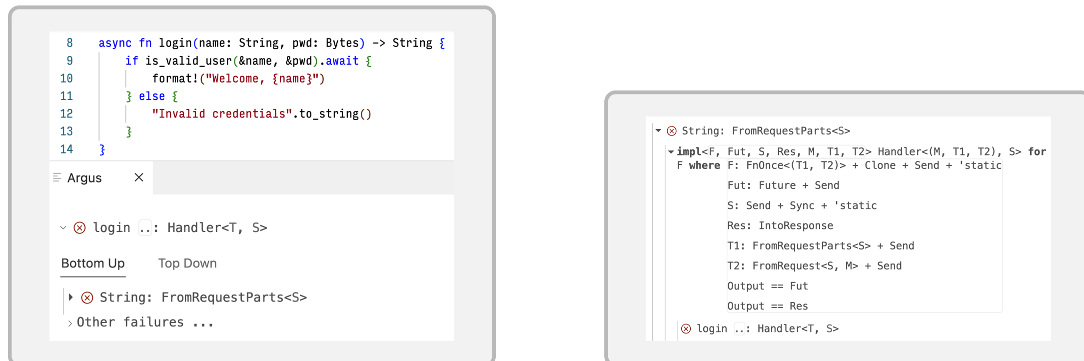


Figure 3.3: Left: Argus is able to localize the root cause of the failed trait bound, `String: FromRequestParts`, and presents this first in the bottom-up debugging view. Right: Developers can expand this failure up to the root goal to see the full provenance.

root cause. In lieu of reductive diagnostics, we preserve the proof tree and are solving an interface problem. This means Argus doesn't shy away from branching—it embraces the full proof tree—developers can always dig further down into failed trait bounds. We now see Argus in action. The root error cause of code is that `String` doesn't implement `FromRequestParts`, barring the function from implementing the desired `Handler`.

Seen in Figure 3.3, Argus has done something that Rust couldn't—or wouldn't—it has reported the root cause of the trait error. We did not solve the branching problem or deploy sophisticated diagnostic strategies. Argus uses a small set of simple heuristics and filtering to suggest root causes. These heuristics are discussed in Section 3.2. Argus can rank the likelyhood of each error and present them to the user because it *does not reduce* the proof tree, where compilers need to be conservative, Argus can be daring.

The problem of proof tree exploration is that of an interface problem. The compiler has large data, Rust types are large, and yet we need to make this information consumable. Argus isn't tied down to reporting textual static diagnostics, opening the room to interface design experimentation. Some of the ways in which Argus deals with large data is path shortening and trimming trait implementation headers.

PATH SHORTENING The *definition path* of an item is the absolute, qualified path uniquely identifying its definition. Common paths such as `std::vec::Vec` don't seem too harmless. Library paths, especially those involving traits, are a bit harder to read

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

```
use diesel::prelude::*;

table! {
    users(id) {
        id -> Integer,
        name -> Text,
    }
}

table! {
    posts(id) {
        id -> Integer,
        name -> Text,
        user_id -> Integer,
    }
}
```

Figure 3.4: A small media platform with users and posts, application uses the Diesel library to achieve static safety when building queries. The function `users_with_equiv_post_id` returns all users who have a post with an id equivalent to theirs. This query is invalid because the tables need to be joined before filtering, the missing line is commented out.

at times. Consider a simple social media application whose implementation uses the popular Diesel library to handle relational mapping and query building.

The application has two tables, `users` and `posts` , and for some reason there's a query to select all users who have a post with the same id as their user id. (A promotional idea perhaps?) The code shown in Figure 3.4 doesn't compile, there's a missing `INNER JOIN` on the tables before filtering on equivalent IDs . Diesel pushes these errors into the trait system to provide static safety. The fully qualified type in the principle error message is below.

```
<QuerySource as diesel::query_source::AppearsInFromClause<posts::table>>::Count
  == diesel::query_source::peano_numbers::Once
```

To further beat a dead horse Rust reports "the full type name has been written to 'file.long-type.txt"' where all types involved in the error are written. Ponder this statement. The type was *too long* to be displayed and the developer needs to dig through output files if they want to see it.

Temporary debug files have several hashes in their names making the real filename much longer too.

Instead of shying away from these large qualified paths, Argus can shorten them by default, and provide the fully qualified version on hover, demonstrated in Figure 3.5.

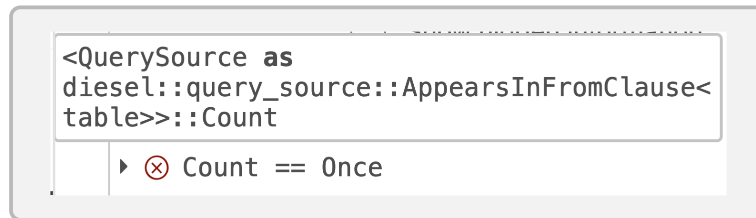


Figure 3.5: Argus provides fully-qualified paths on hover. Shortening paths by default helps the interface remain readable and reduces visiual clutter. Here the user is hovering over the symbol `Count` to see its fully qualified definition path.

Hiding qualified types by default reduces the size of information attacking developer's visual system and helps them sift through information faster. There are, of course, instances where shortening paths by default can be confusing. If multiple paths shorten to the same symbol, for example trait methods within different implementation blocks, information may seem redundant or conflicting. In these cases we provide a small visual prefix to indicate that the symbols are part of a larger path.

TRIMMING IMPLEMENTATION HEADERS Documentation is a crucial resource to help developers, especially when using new libraries. Rust documentation is generally good, but it could do better especially when it comes to traits and their implementors. Consider again our running example with Axum. We have a function `login` that fails to satisfy the criteria to implement the `Handler` trait. The reported error is like "trait bound `login: Handler` is not satisfied." The intuition for many developers is to visit the documentation page for handlers to see what types do implement the trait. Unfortunately, the verbosity of Rust is a little jarring when sifting through dozens of implementation types in documentation.

Shown in Figure 3.6 is the implementation block for functions of arity sixteen. There is a nearly equivalent documentation item for functions of arity fifteen, and fourteen, and thirteen ...ldots and so on down to zero. These blocks are trying to say the same thing: handlers are asynchronous functions whose first $n-1$ arguments implement `FromRequestParts` and whose n^{th} argument implements `FromRequest` and whose return type implements `IntoResponse`. Extra verbosity in the documentation comes from all those type parameters. As previously stated, handlers can have between zero and sixteen arguments, therefore the documentation has one implementation block for each function arity.

Additional Rust-isms like `'static` and `Send` further clutter the documentation.

```
[+]§impl<F, Fut, S, Res, M, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13,      source
  T14, T15, T16> Handler<(M, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
  T15, T16), S> for F
  where
      F: FnOnce(T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16) -> Fut + Clone + Send +
  'static,
      Fut: Future<Output = Res> + Send,
      S: Send + Sync + 'static,
      Res: IntoResponse,
      T1: FromRequestParts<S> + Send,
      T2: FromRequestParts<S> + Send,
      T3: FromRequestParts<S> + Send,
      T4: FromRequestParts<S> + Send,
      T5: FromRequestParts<S> + Send,
      T6: FromRequestParts<S> + Send,
      T7: FromRequestParts<S> + Send,
      T8: FromRequestParts<S> + Send,
      T9: FromRequestParts<S> + Send,
      T10: FromRequestParts<S> + Send,
      T11: FromRequestParts<S> + Send,
      T12: FromRequestParts<S> + Send,
      T13: FromRequestParts<S> + Send,
      T14: FromRequestParts<S> + Send,
      T15: FromRequestParts<S> + Send,
      T16: FromRequest<S, M> + Send,
```

Figure 3.6: Example implementation block from the Axum `Handler` documentation. Implementations with a high number of type parameters become hard to read. An unfortunate pattern in Rust, especially when implementing a trait for functions of varying arity.

Looking through this documentation can be confusing and it isn't initially apparent which block one should start with. Ideally, Rust would have *variadic generics*, a long sought after feature,[1] that would allow all these implementation blocks be written as one. Variadic generics are unlikely to land in the near future so Argus does its best to hide the unnecessary information by default. Shown in Figure 3.7 is how Argus hides type parameter lists and where clauses by default, shifting focus to the type and trait involved. Of course, this information isn't lost and users can click the area to toggle its view as seen in previous figures. A small sample of user feedback suggests that the Argus view is more readable than online documentation. This isn't to say it couldn't be improved, future work will include hyperlinks to library documentation and improved rendering of `Fn` trait output types. Using the notation `-> Ty` instead of `Fn::Output == Ty` to indicate return types.

1 See online discussions at variadic-generics-design-sketch/18974.

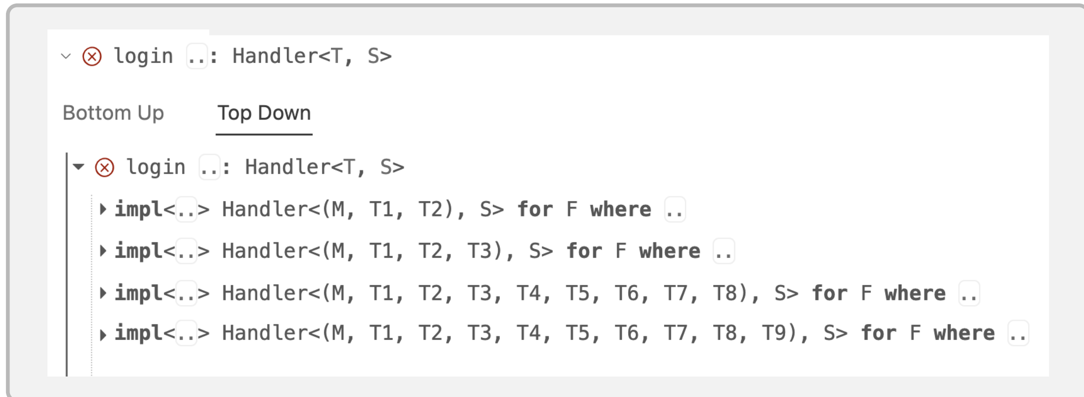


Figure 3.7: Example of how Argus makes implementation blocks more readable. Type parameter lists and where clause constraints are hidden by default, reducing much of the visual noise. Items in the top-down view are sorted by the failure nodes they lead to, here, the block for functions of arity two, `Fn(T1, T2)`, is shown first.

3.2 PULLING A RABBIT OUT OF THE HAT

Compiler writers have for years used a bag of heuristics to provide error diagnostics. The sophistication of these heuristics increase alongside the sophistication of the type system whose errors they are diagnosing. Heuristics are imperfect, and this work shows that interactive debugging uncovers the root cause of errors when static diagnostics cannot. However, Argus also falls victim to the heuristic. It is not sufficient to simply give developers a whole proof tree and say "there, go digging!" We must also tell developers where to start, which failures are most likely the root cause. For this reason Argus comes with its own set of heuristics to rank failed errors. We want developers on the right debugging track with as few interactions as possible.

Providing the exact root failure is undecidable. The Rust trait system is Turing-complete and thanks to our friend *the Halting Problem* we cannot guarantee termination of running a Turing-complete program. Debugging is a *human problem*, not an algorithmic problem. Therefore we cannot expect that Argus, or any tool, be able to accurately localize arbitrary program errors algorithmically. Indeed there are even problems where there doesn't exist only one failure, but a set of failures working together. (See Figure 3.11 for an example.) Argus provides a set of failed obligations in its bottom-up view, and heuristics are used to rank them by their likelyhood of being a root cause.

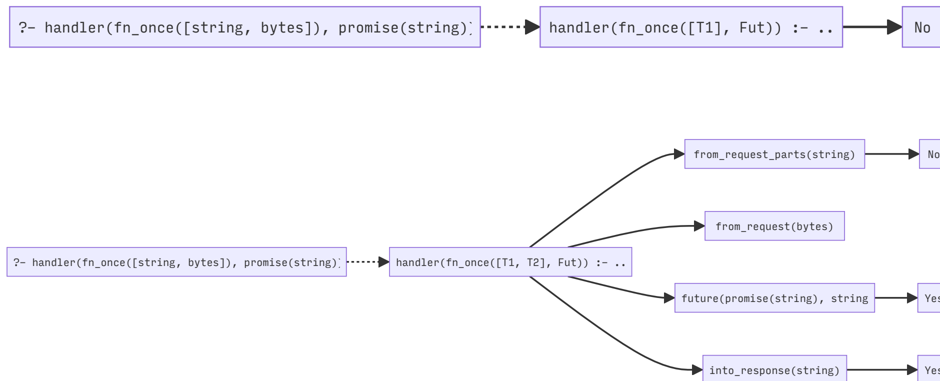


Figure 3.8: Clipping And–Or trees along the disjunctions returns a forest of failed proof trees, each with a direct path of failure from root to leaf. Leaves may contain a conjunctive set of goals that can be analyzed together to understand the nature of the failure.

In this section obligations with potentially useful data are described as *interesting*. Identifying interesting failed obligations is a two step process: conjunctive subgoals are analyzed together as a set—this to first pare down the set of possibilities, then failed leaf obligations are ranked and partitioned on individual merit.

3.2.1 *Conjunct Analysis*

Imagine snipping the And–Or tree along the Or relations. Doing so produces a forest of failed trees . Each tree in the forest has no branching except for a failed conjunctive set at its leaf. See Figure 3.8 for the resulting forest of snipping the proof tree of the Axum `login` example. Each conjunctive set is then analyzed to find those with interesting obligations.

Each tree is a failure otherwise the initial goal wouldn't have failed.

Conjunct analysis uses three metrics to determine whether or not a set contains interesting failed goals. The depth of the failed conjunct, the number of known principle types, and the ratio of conjunct size to the number of known principle types. Conjuncts are interesting if any of these metrics lie more than one standard deviation outside the average for all sets.

DEPTH Trait solving is a *fail fast* process. This means that if there isn't enough information at any point the process stops along its current branch and returns an ambiguous response. Failures deep in the proof tree indicate that the solver was able to go further along the current branch. Progress was made. Intuitively we can think of the solver as being closer to a solution in a deep failure than in a shallow one. This metric helps combat Prolog-style unification failures when many inference variables remain unsolved. Refer back to the running example using the Axum `Handler` trait. The library uses macros to generate the trait implementation rule for functions of arity zero to sixteen. If the function used as a handler has arity two, this means there will be *fourteen* candidates whose rule head does not properly unify with the function type. Unification failures are shallow. The candidate that properly unifies with the rule head, in this case of function arity two, will result in the rule subgoals being solved for—at a greater depth. Appealing to this intuition we conclude the general rule that deep failures are more interesting than shallow ones.

NUMBER OF KNOWN PRINCIPLE TYPES A trait goal of the form `Type: Trait` has principle type: `Type`. Of course obligations have a left-hand-side type, why make this distinction? Trait solving in Rust is interleaved with type inference. Many goals may start looking like `0?: Trait`, where `0?` is an *inference variable*. Its type is currently unknown, referring back to the goal, this goal has an unknown principle type. These goals are often answered with 'ambiguous' by the trait solver, but may produce a set of additional constraints on the inference variable. We may learn that `0? = Vec<1?>`, meaning the inference variable is a vector whose type parameter remains unknown. Note that the goal `Vec<1?>: Trait` has a known principle type. As a metric we favor groups that have a higher number of known principle types. They are more interesting than groups with remaining inference variables, which are bound to fail by construction.

RATIO OF SIZE BY KNOWN PRINCIPLE TYPES Again using the number of known principle types we can additionally take into account the group size. Let us say there exist two groups, each with ten obligations with known principle types. Is one more interesting than the other if the groups have sizes 10 and 100? We argue yes. Small groups with a high number of known principle types are more interesting. Their data is more concrete. Failures are more concrete. This leads to the third metric, the ratio of group size by the number of known principle types; this favors smaller groups with more concrete information to aid debugging.

The proposed list of metrics is by no means exhaustive and will likely be extended in the future to analyze conjunctive sets more thoroughly. See Chapter 4 for ideas lifted from related research areas. Using this list we filter out conjunctive sets that do not meet our criteria. Now, failed goals are ranked amongst themselves to lift interesting failures to the top of the Argus proposed list.

3.2.2 *Subgoal Analysis*

Analyzing sets of failed goals together is how we catch all interesting obligations. This does not mean that all members of an interesting set are themselves interesting. To throw out these wolves in sheep's clothing we partition the set of interesting groups one more time, throwing out individual obligations that are uninteresting.

In this context an interesting obligation is one that provides concrete information. This happens in one of two ways, either its principle type is known, or the obligation resulted in "no" or "maybe: overflow." Ambiguous obligations with unknown principle types are automatically thrown out. They provide the least amount of debugging help. This naïve partion is sufficient to remove obligations that produce visual noise and little useful information. They are however still available in the bottom-up debugging list, just at a lower rank.

3.3 system design

The chiseled statue presented in research was, just a few months prior, a heaping mess of granite. The process and challenges in a project are often the most interesting technical aspects, this section highlights specific design decisions and rough edges of the Argus internals. We finish with a discussion on known shortcomings and future plans.

Shown in Figure 3.9 is the general system architecture of Argus. There is a main component, `argus::analysis`, that type-checks a Rust module and records all obligations solved for by the trait solver. These obligations are available via the `inspect_typeck` [2] function that exposes obligations via a callback. This feature was made available by PR 119613 to expose obligations such that third party tools can extract generated proof trees. After type-checking, Argus analyzes and filters the recorded obligations in the

2 https://doc.rust-lang.org/stable/nightly-rustc/rustc_hir_typeck/fn.inspect_typeck.html

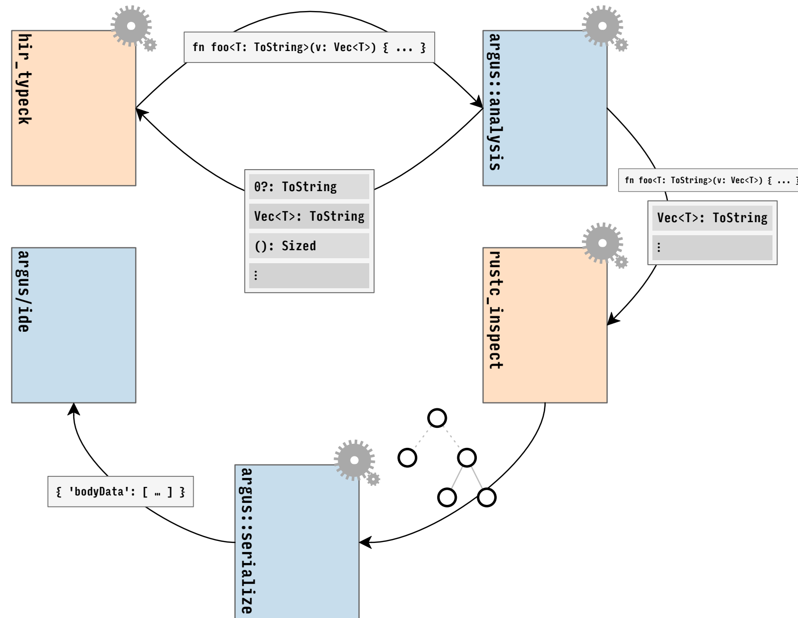


Figure 3.9: Abstracted architecture design of the Argus system. Orange boxes are Rust compiler modules and blue boxes are Argus modules. `argus::analysis` is the system entrypoint that passes Rust HIR items, such as functions, to be type-checked. While type-checking a body, Rust invokes an Argus callback passing solved obligations and results. After type-check Argus processes and filters these obligations then extracts the proof tree from the trait solver via `rustc_inspect`. Information is then serialized into JSON and passed to the web view.

`argus::analysis` module. Analysis results are passed back to the Rust compiler to extract necessary proof trees via the the proof tree visitor API. All data is then serialized into a custom JSON format and passed to the Argus web view. Here we detail the filter and analysis steps of the `argus::analysis` module.

Argus has little dependence on VSCode. There is even an Argus MdBook utility.

3.3.1 *You Want What Again?*

The first step in the Argus pipeline is to type-check the workspace modules and record obligations solved for during trait resolution. Post initial type-checking, Argus now has a large set of obligations and needs to determine which the developer would find useful. This set is comprised of *all queries* made by the trait solver, determining which are useful to debug an error is difficult for three reasons. First, we don't want to expose overly complex compiler internals. The trait system has several variants of obligation, many of which, don't have a source code representation. Second, after a failed obligation the trait solver may learn new information and then retry the query. There is no direct mechanism to distinguish a failed obligation from a failed obligation that succeeds on a subsequent solve attempt. Third, the trait solver can solve obligations that are not hard requirements, but rather, the answer informs type inference. I will refer to these as *softline* obligations, their failure does not always imply a type-check error.

COMPLEX OBLIGATIONS Until now, discussions of obligations centered around the form `U: T`, read as: "does type `U` implement trait `T`?" As of April 2024, Rust has *eight* kinds of predicates the trait solver can ask, and an additional seven "clause kinds" (a kind of predicate). Only *three* in total have an analogous source-code representation. By default Argus only shows these three clause kinds to users:

- trait predicates: `U: T`
- region outlives predicates: `'a: 'b`
- type outlives predicates: `U: 'a`

Other predicates are hidden by default and shown only if they are the direct cause of a type-check error. This includes queries such as "Is type `U` well-formed?" and "Can trait `T` be used as a trait object?" The goal is to avoid overwhelming developers with information they may not need. (Or may not have known existed!)

SUBSEQUENT OBLIGATIONS The trait solver not only returns a yes/no/maybe answer for each obligation, but also a set of inference variable constraints. Added inference constraints may cause a previously ambiguous obligation to succeed. Consider again the simple `ToString` trait example. The type signature of the `print_ln` function is `fn print_ln<T>(v: T)where T: ToString`, this means that the caller needs to provide a type `T` that satisfies the bound `ToString`. In the calling context the first obligation given to the trait solver is `0?: ToString`, where `0?` is an inference variable. This obligation will result in an ambiguous response. However, the solver will add the inference constraint `0? = Vec<f32>`. The subsequent iteration of this goal, `Vec<f32>: ToString`, is not ambiguous and fails. We do not want to show developers the first ambiguous obligation, only the second failure. Argus currently piggybacks on Rust compiler infrastructure to test whether a failed obligation O_F implies an ambiguous obligation O_A. It should be noted that this strategy is unsafe due to inference variables potentially leaking out of context.

SOFTLINE OBLIGATIONS Obligations received from the compiler come marked with a "cause code." This encodes data about the origin of the obligation. For example, types returned from functions need to be sized. Consider the function `fn to_string(&self)-> String` from the `ToString` trait. The obligation `String: Sized` is solved for with a cause code `SizedReturnType`.[3] Unfortunately, to the authors chagrin, there exists a dummy cause code, `MiscObligation`[4] that can be used when the obligation is ill-classified. This cause code is also used for softline obligations. The issue is that Argus cannot ignore these obligations as they can demonstrate important failures.

Consider again the `ToString` trait, we will modify the example `main` function from Section 1.2 to that shown in Figure 3.3.1.

The Rust compiler reports the correct root cause, but it attempts solving an additional obligation while searching for an appropriate trait method implementation . In this case the miscellaneous obligation `Vec<A>: Display` is solved for.

The default behavior for Argus is to ignore miscellaneous obligations, unless their span can be tracked to a method call. For method calls, Argus will remove miscellaneous obligations and re-perform method selection, registering each obligation as required.

3 https://doc.rust-lang.org/stable/nightly-rustc/rustc_infer/traits/enum.ObligationCauseCode.html#variant.SizedReturnType
4 https://doc.rust-lang.org/stable/nightly-rustc/rustc_infer/traits/enum.ObligationCauseCode.html#variant.MiscObligation

```
struct A;

fn main() {
  let v = vec![A];
  format!("{}", v.to_string());
}
```

Figure 3.10: A modification of the running example trait `ToString`. The struct `A` is introduced and it implements no trait, therefore, the method call `v.to_string()` is ambiguous and the code fails to type-check. In addition to `Vec<A>: ToString`, the Rust compiler also solves for `Vec<A>: std::fmt::Display` with a miscellaneous cause code. The compiler tries this additional obligation in an attempt to provide a helpful diagnostic to the developer.

3.3.2 *Moving Forward*

In future releases of Argus we hope to improve the collect and filter steps in `argus::analysis`. In certain circumstances Argus filters too aggressively and important obligations are filtered from the default view. This can be improved by tracking the cause code within the Rust compiler more precisely, or a smarter analysis to distinguish truly miscellaneous obligations from important ones. Memory consumption during obligation collection can spike in even medium-sized codebases. Argus needs to maintain information about each obligation that it wishes to analyze further, especially ones for which a proof tree is generated. The current Argus release does little to throw away data it will not use and these extra obligations take up space. To throw away unnecessary obligations again requires the knowledge that an obligation will not be shown to the user, our hope is that the problems during collect and filter can both be mitigated by the extra provenance information coming from the compiler.

3.4 EVALUATION

The thesis of this work is that interactive debugging facilitates localization with *few* interactions. Users can explore the proof tree either top-down, from the root, or bottom-up, from failed obligations. An error is *always discoverable* by doing a full traversal of the top-down tree—this however does not make a quality debugging experience. Discussed in Section 3.2 are the heuristics Argus employs to rank failed obligations in

the bottom-up view. We want users to encounter the root cause as soon as possible. This work evaluates Argus on two research questions:

RQ1 How many cognitive steps does it take to reach the root cause in the top-down and bottom-up views?

RQ2 Does our ranking heuristic reduce the number of cognitive steps in the bottom-up view?

3.4.1 *Procedure*

A 2022 Rust Foundation grant tasked a member of the community to improve trait error diagnostics. The solution proposed in this work is discussed later, see Chapter 4, but the grant awardees assembled a mass of difficult-to-debug trait-error-laden programs from the community—in other words, an Argus test.[5] This suite relies on the trait-heavy crates, such as the previously mentioned Axum and Diesel. Table 3.1 shows the full list of included crates.

It is interesting that many of the mentioned libraries boast being "simple," or "for humans." Yet their usage obfuscates diagnostics and make debugging difficult.

The notion of a root cause is important for our evaluation. Each test was analyzed and the root cause was predetermined by the authors. In some cases tests from the community suite were split and modified to contain exactly one error. After analyzing all the community tests several were withdrawn from the Argus suite for one of three reasons. Either the root cause remained ambiguous after test simplification, the behavior of the stable and in-progress trait solver diverged, or a bug within Argus caused evaluation to fail.

3.4.1.1 *Ambiguous root causes*

Discussion of a root cause until now has assumed that there exists a single root cause. However, by the nature of some trait errors it may not be possible to present a single failed obligation to a user. To explain how this might happen let us turn our attention once again to the Axum web framework. Recall that in our running example the handler function did not implement the trait `Handler` due to an incorrect parameter type. This example reflects the type of trait error with a root cause, reflecting the kinds of errors

5 `https://github.com/weiznich/rust-foundation-community-grant`

CRATE	DESCRIPTION
uom	Units of measurement
typed-builder	Compile-time type-checked builder derive
easy-ml	Machine learning library providing matrices, named tensors, linear algebra and automatic differentiation aimed at being *easy to use*
diesel	A safe, extensible ORM and Query Builder for PostgreSQL, SQLite, and MySQL
chumsky	A parser library *for humans* with powerful error recovery
bevy	A *refreshingly simple* data-driven game engine and app framework
axum	Web framework that focuses on *ergonomics* and modularity
entrait	Loosely coupled Rust application design *made easy*

Table 3.1: Crates used in the testing suite of hard-to-debug trait errors. Many emphasize ease-of-use an ergonomics but complex trait systems can obfuscate errors when things go wrong.

we can evaluate within the scope of this work. Shown in Figure 3.11 is example code that does not produce a single root cause failure.

In Figure 3.11 the developer attempts to use `struct A` as a handler. This struct doesn't satisfy any of the handler requirements and therefore its usage results in an error. Note that this error doesn't have a specific failure. `A` is not a function, regardless of arity. `A` does not implement `IntoResponse`. It isn't that a single obligation fails, but rather they *all* fail.

If a value can be turned directly into a response it is usable as a sort of static handler. A subtle detail that hasn't been important for previous examples.

Argus reports the error list shown in Figure 3.12. It is the sum of all these failed obligations that describe the failure, `A` is incompatible as a handler in every way possible. From the initial test suite six test cases were removed due to an ambiguous root cause.

3.4.1.2 *Divergent Errors*

Argus requires a nightly version of Rust with the in-progress trait solver enabled. This solver still has known unsoundness and incompleteness issues. Any test case within the suite that resulted in a different trait error, or produced an incorrect result by the

```
use axum::{body::Bytes, routing::post, Router};
use tokio::net::TcpListener;

struct A;

#[tokio::main]
async fn main() {
  let app = Router::new().route("/login", post(A));
  let listener = TcpListener::bind("0.0.0.0:3000")
  .await.unwrap();
  axum::serve(listener, app).await.unwrap();
}
```

Figure 3.11: Example program where multiple failed goals combine to form the root cause. The struct `A` is neither an asynchronous function nor does it implement `IntoResponse`. In terms of the logic model, all program clause heads fail to unify with the goal. Developers are left to look at all failed obligations to understand the failure.

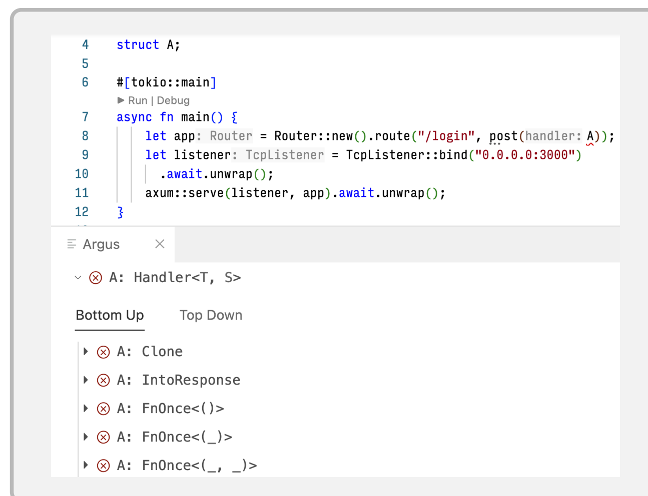


Figure 3.12: First few entries of the Argus bottom-up list view for the "not a function" program. An example error where Argus reports all the correct information, though it lacks concrete information for developers. The struct `A` does not satisfy any of the `Handler` trait constraints. Developers must currently intuit the failure from the sum of all failed candidates.

definition of the Rust semantics, was thrown out. From the initial test suite only two tests were removed for this reason.

3.4.1.3 *Argus Bugs*

Understatement of the century?

Despite what they authors want to believe, even research software can contain bugs. Two tests from the original community suite were removed due to a bug in the Argus implementation. It should be noted that neither of these pose threats to the validity of the software but highlight engineering difficulties. (Notably in serialization.)

After splitting and filtering the community suite, there remained 12 test cases on which we ran the evaluation. The root causes for each test were determined by the authors and written in plain text in the Argus test suite. We use an automated test runner to load each workspace into a Playwright chromium instance where Argus is opened. Our tool expands the bottom up and top-down versions of the proof tree views and textually searches for the root cause. Nodes in the bottom-up view are evaluated on *rank*, i.e., their position in the list. Nodes in the top-down view are evaluated by the number of "cognitive steps" away from the tree node they are. We assume that developers know exactly which tree nodes to expand and read strictly from top-to-bottom. This measure is therefore a lower bound for the top-down debugging process. We define the cognitive steps a developer would have to take as reading or clicking on a node in the proof tree. The nodes in the top-down view were manually checked by the authors to ensure a correct count of cognitive steps.

3.4.2 *Results*

Figure 3.13 shows the cumulative distribution of the fraction of tasks for the bottom-up and top-down metrics. Left is the fraction of tests such that the root cause was at rank K. Remember that K is the position in the bottom-up list. Right in the figure is the fraction of tests such that the root cause was S cognitive steps from the top-down tree root.

Figure 3.14 the maroon line shows the cumulative distribution of the fraction of tests such that the root cause was at rank K when shuffled randomly. Shown in blue for contrast is the CDF when nodes were ranked by the Argus heuristic.

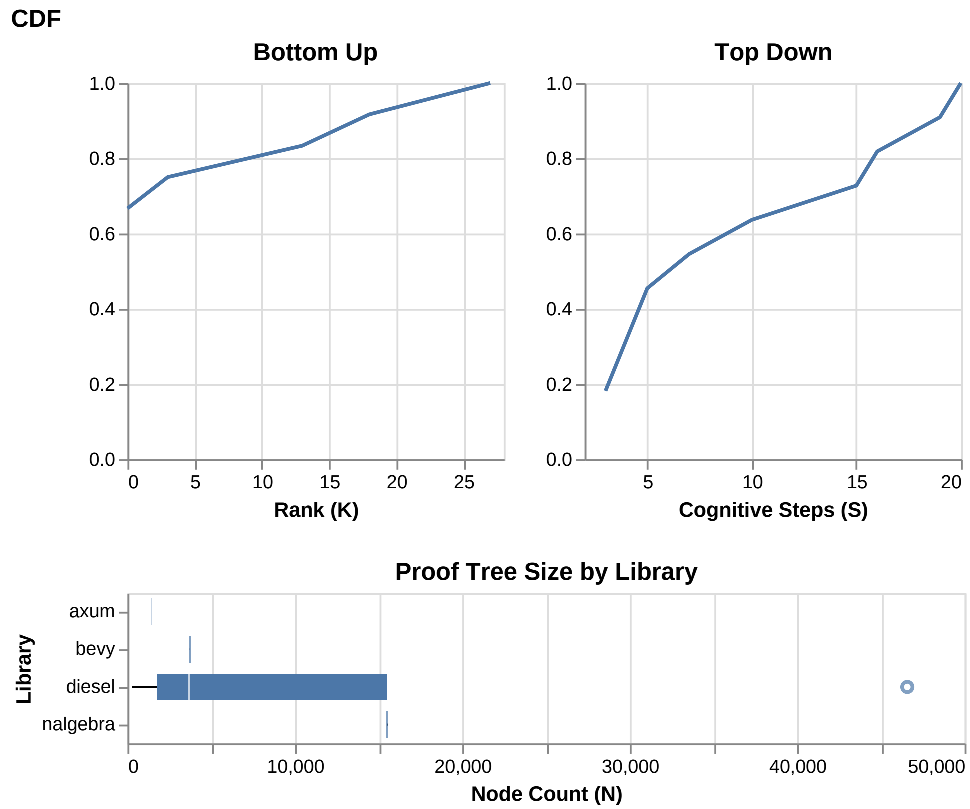


Figure 3.13: Top: the cumulative distribution function for the fraction of test below a threshold. Left: Fraction of tests with a root cause rank K. Right: Fraction of tests with a root cause cognitive steps SK away from the tree root. Bottom: Size of tested proof trees by library.

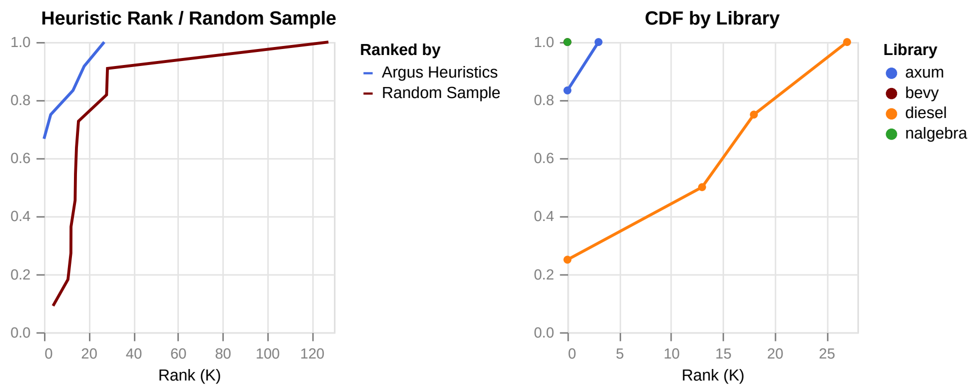


Figure 3.14: Cumulative distribution of the fraction of tests whose root cause was ranked K. Left: Shown in blue, the distribution of errors ranked by the Argus heuristic. Contrasted in maroon is the distribution of errors averaged from a random sample of size 20. Right: Cumulative distribution by library involved under test. Argus heuristics perform notably worse with Diesel than other trait-heavy libraries. (*Note: the Bevy and nAlgebra cases overlap in the top-left corner.*)

3.4.3 *Analysis*

RQ1 The comparison between bottom-up and top-down nodes shows a substantive improvement in the number of interactions required before encountering the root cause. This is supported by the assumption made about the cognitive steps to reach an error node. We've assumed that the developer will always make the right decision when expanding the tree, making S a lower bound on the cognitive interactions required. In practice, the proof trees can have a high branching factor and developers unfamiliar with the library internals could easily expand subtrees in the wrong direction. We point readers to Figure 3.13 (bottom) where the size of proof tree is shown by library. Axum has the fewest nodes per tree on average, 1758, which is still a large amount of data in which to potentially get lost. The bottom-up is crucial to get developers the information they need faster.

RQ2 The heuristic ablation data, Figure 3.14 (left), show that randomly sorting the error nodes in the bottom-up list is insufficient. The Argus heuristics do, in fact, matter and additional 60% of test cases had the root cause shown first to the user.

This is not to say there isn't room for improvement. Comparing the number of interface interactions between the bottom-up versus top-down we see that in some cases it is easier to traverse the tree. Again, the cognitive steps is a lower bound, but the data highlights that there is room for improvement in the heuristics. The cumulative distribution by library supports the need for a more sophisticated analysis. Tests with Diesel perform much worse than those involving Axum, Bevy, or nAlgebra. Further investigation is required to understand the discrepancy. One intuition may be the high variance in proof tree size, Figure 3.13 shows a node count range 250–47000 for Diesel test cases.

4 RELATED WORK

This work has outlined the difficulty in debugging type class errors. A solution was explored in the context of the Rust Programming Langugae with the intention of making these errors more explorable, and ultimately, easier repaired. However this work is not about Rust, nor is it specifically about type classes. The lens of type classes is used to explore the area of debugging complex logic structures, which are ever more prevalent in modern programming languages. This larger problem encompases many research areas with similar goals: type inference diagnostics, logic program debuggers, proof tree debugging, and the human factors of debugging. These areas should be able to pull ideas from each other to help human developers better understand failing logical mechanisms.

4.1 DIAGNOSING TYPE ERRORS

Hindley-Milner type inference is at once a great triumph of functional programming, and simultaneously a source of unending pain. For 40 years, researchers have proposed increasingly sophisticated methods for diagnosing type inference errors (although none have made it to production, to our knowledge). A variety of strategies have emerged:

4.1.1 *Fault Localization*

As a result of the global nature of HM type constraints, where the type-checker notices an error is not usually where the real fault lies. The strategy of fault localization is to try and blame the "right" line of code for a type error.

Wand [27] developed an algorithm to track the provenance of unifications made by the type-checker, which could then be presented to the user. Wand provides the intuition for needing this information as a question. "Why did the checker think it was dealing with a list here?" Having access to the provenance of unifications, experts can

track the origins of unexpected checker states—the origin of an inconsistent type is likely not intended by the developer.

The usage of the raw proof tree presented by Argus follows the same intuitional logic. Like Wand, we currently rely on the intuition of expert programmers to justify the usefulness of such provenance. A planned extension of our work is to understand what user-friendly or expert-system interactions are desired—for both expert and beginner programmers.

Hage and Heeren [8] developed the Top framework to support customizable type-inference. Traditional HM systems solve constraints as they go, biasing the compiler to report errors towards the end of a program. The Top framework builds *type graphs*, allowing for the use of heuristics over the global type graph to aid error diagnosis. An example heuristic is a "trust factor," which sorts constranits from least to most trusted when placing blame. For example, types imported from the language standard are considered *high-trust*—such functions can only be *used* incorrectly. Argus benefits from such a rule, and even stronger, Rust requires type declarations on all functions, which can be trusted more than inferred types. For a thorough treatment of the heuristics used in Argus see section Section 3.2.

Described in Section 3.1, oftentimes a subtree fails because of many conjuncts. However, one piece of information may make the entire subtree succeed. This same principle applies in fault localization. Many recent systems look for sets of constraints, that if resolved, the program would type-check. Pavlinovic, King, and Wies [14] and Loncaric et al. [13] use an SMT solver, and Zhang et al. [28] use a Bayesian analysis. Seidel et al. [18] go further and use machine learning to predict blame based on a training set of ill-typed programs.

The Argus heuristics use a similar approach by analyzing conjunctions for certain properties. However we do not look at sets of constraints as final. The obligations in a proof tree deviate from a constraint-satisfaction model because they can be *ambiguous*. Ambiguous obligations, if resolved, could result in further required obligations for the success of the subtree—additional obligations could be unsatisfiable or unintended by the user. A concrete example of this situation was already demonstrated in Figure 3.11 Compiler developers have operated for many years using a heuristic-based approach to diagnostics. While fault localization techniques can help with diagnosing Rust trait errors, they are unlikely to generalize to all cases. Our experience with Argus corroborates this intuition as each heuristic performs different on different shapes of trait system. Fault localization is usefull, not to determine accurate blame, but to point users to likely points of failure.

4.1.2 *Interactive Debuggers*

In contrast to fault localization, which attempts to find the root cause of an error, an alternative is to give the programmer an interface into *all* the information. Chitil [3] designed a *compositional* explanation graph of principal typings. This use of "compositional" requires that the explanation of a problem have a tree structure, where the type of each node must be uniqely determined by the node's children's types. This graph is combined with a command-line interface that asks users about the intended types of expressions in a process called *algorithmic debugging*, introduced by Shapiro [20]. The value of algorithmic debugging is to bring the developer in the loop to understand the mental under which they're work.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ x

last xs = head (reverse xs)
init = reverse . tail . reverse

rotateR xs = last xs : init xs
```

The algorithmic debugging process for this program c To help the tool place blame a series of questions are posed to the user. Below is an example user-interaction script from the above ill-typed program, with the user responses (y/n) appearing to the right.

```
Type error in: (last xs) : (init xs)

last :: [[a]] -> a
Is intended type an instance? (y/n) n

head :: [a] -> a
Is intended type an instance? (y/n) y

reverse :: [[a]] -> [a]
Is intended type an instance? (y/n) n

(++) :: [a] -> [a] -> [a]
Is intended type an instance? (y/n) y
```

Using this script the tool can identify that the error lies within the function `reverse`, but the inferred type of the append operator `++` is not to be blamed.

Algorithmic debugging is explained in great detail because it provides a promising direction forward for Argus when the tool faces ambiguity. In scenarios of great ambiguity such as Figure 3.11 human input can help Argus provide better feedback.

Furthermore, interactive debugging in the context of typing has been refined in multiple ways. Tsushima and Asai [25] developed a tool to work with the OCaml compiler rather than requiring a "debugger-friendly" re-implementation of type inference. Similar to some fault localization approaches, Stuckey, Sulzmann, and Wazny [24] uses constraint satisfiability and constraint provenance as a debugging aid. Chen and Erwig [2] use a "guided debugging" approach with *counter-factual* typing. A comprehensive set of type-change suggestions is filtered and presented to the programmer to find correct code changes [1].

Our approach to type class debugging took inspiration from many of the above tools. It distinguishes itself as being a richer 2D graphical interface that integrates into the existing VSCode IDE. This integration is essential to map errors to respective source locations, and does not require users to exit their development environment to run a command-line tool.

The published version does not provide an algorithmic debugging interface to refine error suggestions nor does it use counter-factual typing to propose code changes. An algorithmic debugging interface would certainly be a simple extension to the Argus system and rule out many "trivially false" obligations with a few user questions.

Code suggestions is an oft-requested feature. Understanding the source of an error is great, but wouldn't it be better to "magically" provide a solution too? This is indeed a motivation behind program synthesis and automated repair.

4.1.3 *Automated Repair*

Stronger than the counter-factual typing presented in the last section by Chen and Erwig [2], several systems attempt to identify a small change to the input program that causes it to be well-typed [1, 12, 17]. In this thesis the proposed method of type class debugging aims to exhaust other avenues for debugging before reaching to program synthesis—a sledgehammer tool. A direction for exploration would be to present users with a set of *types* that do satisfy a class bound. Indeed the Rust compiler already attempts to do this under certain circumstances. However, Rust will not compose types for these suggestions, but following the obligations of a failed proof tree provides a recipe with which a tool *could* compose types to satisfy a given bound. This set of types is potentially empty if no instances are declared or are not visible at the error location. This feature would be more lightweight than full-blown synthesis as only the types are suggested, not an instance of those types.

4.1.4 *Domain-specific annotations.*

evidenced by the myriad of techniques to generally diagnose type errors, building a fully generic diagnostic system is a tricky task. An alternative approach is to give library authors the necessary tooling to inject domain-specific knowledge into diagnostics.

The Helium subset of Haskell introduced by Heeren, Hage, and Swierstra [9] took such an approach. Library authors can use a DSL to express custom errors messages that reflect common errors known to library maintainers.

Notably, most of the efforts in the Rust ecosystem towards addressing trait errors also have the shape of domain-specific annotations. RFC #2397 [1] describes a `#[do_not_recommend]` annotation that library authors could place on certain trait implementations. For instance, if a trait is implemented for tuples of length 32, then a library author could mark that implementation to not appear in the suggestions of diagnostics.

At time of publishing, the Rust compiler includes a nightly feature allowing library authors to append additional notes to diagnostic messages. One good example of its usage is in the Axum web framework. The pitfalls of creating a correct `Handler` have been discussed at lenghts in this written work. A portion of RFC #2397 was to improve error messages and here the following annotation was included on the Handler trait:

```
#[cfg_attr(
  nightly_error_messages,
  rustc_on_unimplemented(
    note = ''Consider using '#[axum::debug_handler]' to improve the error message''
  )
)]
```

This annotation framework is not as expressive as Helium's DSL so the authors have created a custom macro that does the heavy lifting, `axum::debug_handler`. This macro expands into additional code that does a static type assertion for all the Handler constraints. This macro-based approach is convincing and in many cases provides the current most-specific diagnostic. The downside is every library needs to create, and maintain, these additional debugging facilities. One attractive feature of the Helium system is that error notes are statically-checked to ensure compatibility with the current function types.

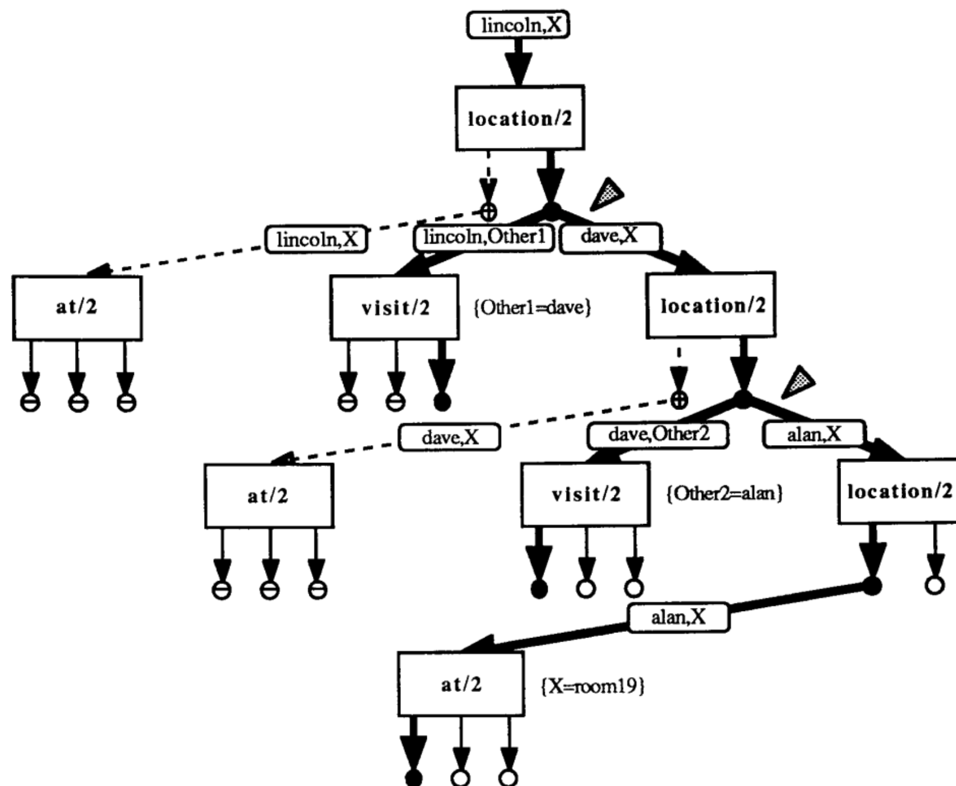


Figure 4.1: Example logic program trace visualization. A cyclic And–Or tree from Senay and Lazzeri [19].

4.2 LOGIC PROGRAMMING

While logic programming has fallen out of fashion, the sizable research program around it in the 1980s and 90s has left us many interesting threads to potentially pick back up. In particular, researchers developed a number of tools to facilitate debugging of Prolog programs. Several systems visualized And–Or trees that represented the execution trace of a Prolog program: the Dewlap debugger [4], the Transparent Prolog Machine [6], and cyclic And–Or graphs [19]. Figure 4.1 shows an example diagram from the last system. Other Prolog debuggers like Opium [5] focused on abstracting data and control-flow within large execution traces.

These trees provide some guidance in how to compactly visualize various aspects of a logic program trace, such as unification and backtracking. The Argus system adopts the And–Or structure of the tree but puts a larger emphasis on scaling. Large-scale visualization become illegible quickly and we need to make sure developers can find the information they're looking for as quick as possible. The idea is that with a quick scan they already have a better idea of what went wrong.

For example, the AORTA diagrams from Eisenstadt include small icons on the nodes and edges that encode control flow direction and unification through the tree. These icons are tremendously helpful in the given example (Figure 4.1) but do not scale well for larger problems. Argus puts a much larger emphasis on *hiding* unnecessary information by default, but giving developers the settings to opt-in should they choose. Later work on the Transparent Prolog Machine emphasized helping users orient themselves while inspecting a large tree, and some techniques may be appropriate for future work.

4.3 DEBUGGING PROOF ASSISTANTS

The use of tactics in modern proof assistants seems to have the same flavor of usability problem as trait errors. For example, a programmer applies a tactic to some goal; the application either succeeds or gets simply a "No" failure. In theory, similar techniques might be useful to diagnose a trait error as to diagnose a tactic failure. However, we struggled to find much related work on this subject. In the space of proof assistants there are two styles of tool, proof visualizers and proof *state* visualizers. The former

1 rust-lang/rfcs/2397-do-not-recommend.md

emphasizes writing proofs, following a pattern similar to that taken on paper, and the latter emphasizes understanding the state of the current proof.

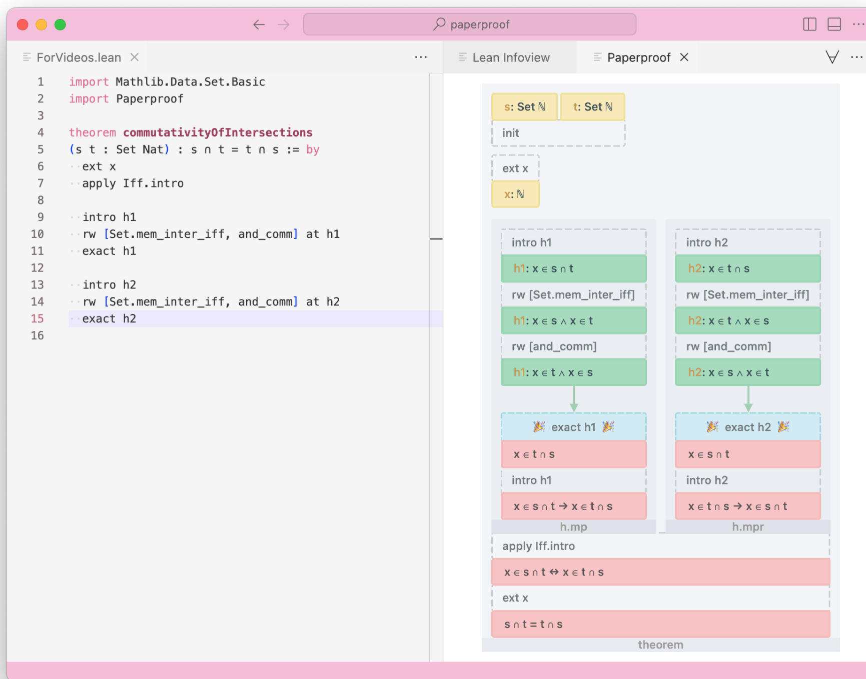
Proof state visualizers render the proof tree *before* and *after* each tactic application, but do not help with understanding the application itself. Notable tools in the space include Alectryon [16] for Lean and Coq-PSV [7], and Shi, Pierce, and Head [21] describe a "tactic preview" which can help readers of proofs identify the goals solved by a given tactic. Even though proof state visualization may seem tangential to trait solving, an extension for Argus is to iteratively step through the solver's execution visualizing the type state at each step. We imagine this feature would be more useful to compiler writers and library maintainers.

Proof visualization tools seem to take an approach of modeling sequent calculus with Gentzen trees. These tools aim to help proof writers by providing a more "paper-like" feel to mechanized proofs. The oldest tool in this space, ProofTree, works with Coq and ProofGeneral and displays the tree top-down similar to the Argus And–Or tree. Figure 4.2 show two additional tools in the space, Paperproof and Traf [10, 11] that use a more Gentzen-style natural deduction proof to help proof writers.

4.4 FINAL WORDS

Heuristic compiler diagnostics cannot always localize type class errors; interactive visualization of type class resolution is feasible and with few interactions facilitates localization. In Chapter 2 we talked about the reductive nature of compiler diagnostics. Type class resolution is a complex process. In most compilers it is black-box to developers, thus the curt response "No" is a frustrating and hard-to-debug experience. Compilers may use the words "bounds unsatisfied," or "failed to synthesize instance" but they can carry the same emptiness as a "No." Instead of reducing internal failures to a single diagnostic, why not give developers an interface by which they can explore the problem?

Chapter 3 introduced our first attempt at a solution to this problem. How can we give developers the right tools to facilitate debugging? First they need the data. Diagnostics often remove this data—we keep it. Second they need an interface. Diagnostics produce a static file, much like this PDF document, that's not much of an interface. We built an interactive debugger into the popular VSCode editor, we even took this a step further and built an MdBook plugin for Argus.



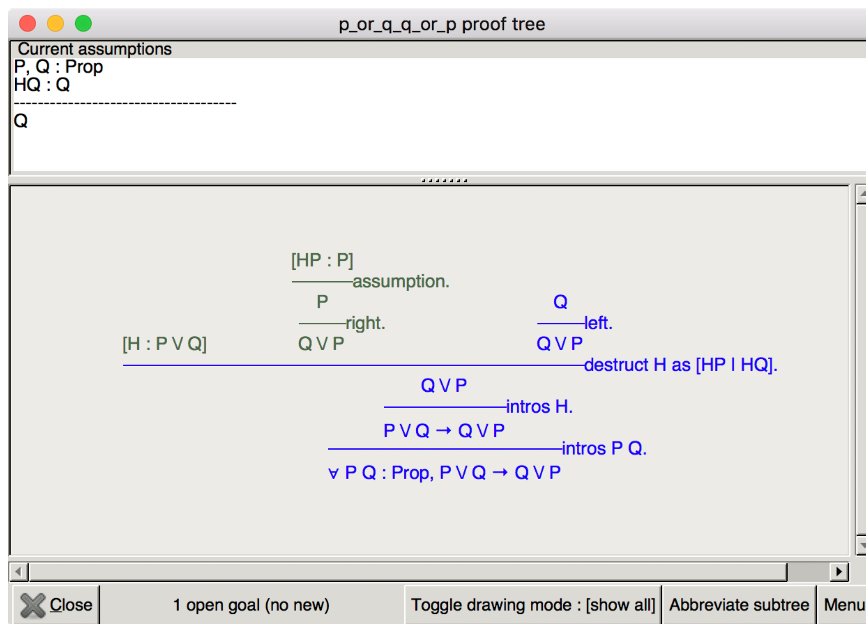


Figure 4.2: Top: The Paperproof VSCode extension interface. Users can interactively build proofs for the Lean language. Bottom: The Traf graphical proof aid for Coq using a Gentzen-style deduction proof.

Visualization comes with inherent problems. Interfaces get visually cluttered, they're confusing to use, and oftentimes too much data can be a bad thing. Especially if that data is leaking parts of the Rust compiler internals. Argus uses some simple heuristics to provide debugging entrypoints to developers. On 80% of our community curated tests these heuristics served the root cause within the first ten items. Ten may seem like a lot, and indeed there's plenty of room for improvement, but often documentation pages contain dozens of items. Not to mention the two or three pages of documentation of required reading before the root cause becomes apparent.

Most researchers would title this a "conclusion," but it's more of an extended aside. The intention was for it to be a discussion, but the fragile web of cross-references did not allow for the already existing "discussion" file to be renamed. Some of the most exciting, frustrating, and interesting pieces of research are not the stuffy words in the text, but rather the software around it. The process. The reference structure around these source files is so fragile because the authoring software used to write the thesis was built while writing the thesis. Turns out this is like building a Formula 1 car mid-grand prix.

The goal was always to have a web version of the thesis, but it turns out universities appreciate a PDF. Well—they require one. The solution was to create a custom markup language to write the thesis and accommodate both output formats. This requirement compounded with haste and inexperience resulted in a funky language. Dynamic scoping? Check. Interfaces? Why yes. Macros? Of course. Ultimately this was a stressful but highly rewarding process resulting in a language no one should ever use again.

Many projects have a difficult technical bit that no one cares about. You can shout *I spent months working out the details!* Yet it receives little interest from advisors or peers. The unsung hero of this story is pretty printing. Section 3.3 contains a loose architecture diagram of Argus. The module `argus::serialize` gets its own box for good reason. This module contains code to serialize the entirety of Rust's type system, taking heavy inspiration from the `rustc_middle::ty::print::pretty` module. If pretty printing doesn't seem difficult, just count the number of `TODO`, `FIXME`, and "uhhh, is this right?" comments in the Rust pretty printing source code. The challenge was worth it. We can now serialize Rust types to JSON and play around with them in Argus.

This work has many possible extensions, and have been outlined in Section 3.3, Section 3.4, and Chapter 4. The future for Argus is hopefully upward, and coming to an IDE near you.

BIBLIOGRAPHY

[1] Sheng Chen and Martin Erwig. "Counter-Factual Typing for Debugging Type Errors." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 583–594. ISBN: 9781450325448. DOI: 10.1145/2535838.2535863. URL: https://doi.org/10.1145/2535838.2535863.

[2] Sheng Chen and Martin Erwig. "Guided Type Debugging." In: *Functional and Logic Programming*. Ed. by Michael Codish and Eijiro Sumii. Cham: Springer International Publishing, 2014, pp. 35–51. ISBN: 978-3-319-07151-0.

[3] Olaf Chitil. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors." In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. Florence, Italy: Association for Computing Machinery, 2001, pp. 193–204. ISBN: 1581134150. DOI: 10.1145/507635.507659. URL: https://doi.org/10.1145/507635.507659.

[4] Alan D. Dewar and John G. Cleary. "Graphical display of complex information within a Prolog debugger." In: *International Journal of Man-Machine Studies* 25.5 (1986), pp. 503–521. ISSN: 0020-7373. DOI: https://doi.org/10.1016/S0020-7373(86)80020-7. URL: https://www.sciencedirect.com/science/article/pii/S0020737386800207.

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

[6] Marc Eisenstadt and Mike Brayshaw. "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming." In: *The Journal of Logic Programming* 5.4 (1988), pp. 277–342.

[7] Mario Frank. *The Coq Proof Script Visualiser (coq-psv)*. 2021. arXiv: 2101.07761 [cs.LO].

[8] Jurriaan Hage and Bastiaan Heeren. "Heuristics for Type Error Discovery and Recovery." In: *Implementation and Application of Functional Languages*. Ed. by Zoltán Horváth, Viktória Zsók, and Andrew Butterfield. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 199–216. ISBN: 978-3-540-74130-5.

[9] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. "Scripting the Type Inference Process." In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. Uppsala, Sweden: Association for Computing Machinery, 2003, pp. 3–13. ISBN: 1581137567. DOI: 10.1145/944705.944707. URL: https://doi.org/10.1145/944705.944707.

[10] Evgenia Karunus and Anton Kovsharov. *Paperproof*. https://github.com/Paper-Proof/paperproof. 2023.

[11] Hideyuki Kawabata. *Traf*. https://github.com/hide-kawabata/traf. 2018.

[12] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. "Searching for Type-Error Messages." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 425–434. ISBN: 9781595936332. DOI: 10.1145/1250734.1250783. URL: https://doi.org/10.1145/1250734.1250783.

[13] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. "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*. OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 781–799. ISBN: 9781450344449. DOI: 10.1145/2983990.2983994. URL: https://doi.org/10.1145/2983990.2983994.

[14] Zvonimir Pavlinovic, Tim King, and Thomas Wies. "Finding Minimum Type Error Sources." In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 525–542. ISBN: 9781450325851. DOI: 10.1145/2660193.2660230. URL: https://doi.org/10.1145/2660193.2660230.

[15] Simon Peyton Jones, Mark Jones, and Erik Meijer. “Type classes: an exploration of the design space.” In: *Haskell workshop*. 1997. URL: https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/.

[16] Clément Pit-Claudel. “Untangling mechanized proofs.” In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 155–174. ISBN: 9781450381765. DOI: 10.1145/3426425.3426940. URL: https://doi.org/10.1145/3426425.3426940.

[17] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. “Type Error Feedback via Analytic Program Repair.” In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 16–30. ISBN: 9781450376136. DOI: 10.1145/3385412.3386005. URL: https://doi.org/10.1145/3385412.3386005.

[18] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. “Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis.” In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017). DOI: 10.1145/3138818. URL: https://doi.org/10.1145/3138818.

[19] H. Senay and S. Lazzeri. “Graphical representation of logic programs and their behaviour.” In: *Proceedings 1991 IEEE Workshop on Visual Languages*. Los Alamitos, CA, USA: IEEE Computer Society, 1991, pp. 25,26,27,28,29,30,31. DOI: 10.1109/WVL.1991.238854. URL: https://doi.ieeecomputersociety.org/10.1109/WVL.1991.238854.

[20] Ehud Yehuda Shapiro. *Algorithmic program debugging*. AAI8221751. USA: Yale University, 1982.

[21] Jessica Shi, Benjamin Pierce, and Andrew Head. “Towards a Science of Interactive Proof Reading.” In: *Proceedings of the 13th Annual Workshop on the Intersection of HCI and PL*. PLATEAU ’23. 2023.

[22] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0262193388.

[23] Christopher Strachey. *Fundamental Concepts in Programming Languages*. Lecture Notes, International Summer School in Computer Programming, Copenhagen. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000. Aug. 1967.

[24] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. "Interactive Type Debugging in Haskell." In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell '03. Uppsala, Sweden: Association for Computing Machinery, 2003, pp. 72–83. ISBN: 1581137583. DOI: 10.1145/871895.871903. URL: https://doi.org/10.1145/871895.871903.

[25] Kanae Tsushima and Kenichi Asai. "An Embedded Type Debugger." In: *Implementation and Application of Functional Languages*. Ed. by Ralf Hinze. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 190–206. ISBN: 978-3-642-41582-1.

[26] P. Wadler and S. Blott. "How to make ad-hoc polymorphism less ad hoc." In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. URL: https://doi.org/10.1145/75277.75283.

[27] Mitchell Wand. "Finding the Source of Type Errors." In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, pp. 38–43. ISBN: 9781450373470. DOI: 10.1145/512644.512648. URL: https://doi.org/10.1145/512644.512648.

[28] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. "Diagnosing Type Errors with Class." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 12–21. ISBN: 9781450334686. DOI: 10.1145/2737924.2738009. URL: https://doi.org/10.1145/2737924.2738009.

[29] *diesel-rs GitHub Discussions*. https://github.com/diesel-rs/diesel/discussions?discussions_q=not+implemented+for. Accessed: 2024-04-24.