Meta-Monomorphizing Specializations

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Achieving zero-cost specialization remains a fundamental challenge in programming language and compiler design. It often necessitates trade-offs between expressive power and type system soundness, as the interaction between conditional compilation and static dispatch can easily lead to unforeseen coherence violations and increased complexity in the formal model. This paper introduces meta-monomorphizing specializations, a novel framework that achieves specialization by repurposing monomorphization through compile-time metaprogramming. Instead of modifying the host compiler, our approach generates meta-monomorphized traits and implementations that encode specialization constraints directly into the type structure, enabling deterministic, coherent dispatch without overlapping instances. We formalize this method for first-order, predicate-based, and higher-ranked polymorphic specialization, also in presence of lifetime parameters. Our evaluation, based on a Rust implementation using only existing macro facilities, demonstrates that meta-monomorphization enables expressive specialization patterns—previously rejected by the compiler—while maintaining full compatibility with standard optimization pipelines. We show that specialization can be realized as a disciplined metaprogramming layer, offering a practical, language-agnostic path to high-performance abstraction. A comprehensive study of public Rust codebases further validates our approach, revealing numerous workarounds that meta-monomorphization can eliminate, leading to more idiomatic and efficient code.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software design engineering; Software and its engineering → Correctness

Keywords and phrases Monomorphization, Specialization, Metaprogramming, Higher-Ranked Polymorphism, Rust, Compile-time Code Generation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material Anonymous supplementary material

1 Introduction

Monomorphization. A principal challenge in high-performance systems programming is achieving zero-cost abstraction through *parametric polymorphism* [18, 76].[1] Grounded in the theoretical frameworks of System F [32, 77, 16] and Hindley-Milner type systems [40, 64], it enables the definition of function abstractions and algebraic data types [55, 89, 9] (ADTs) that operate *uniformly* across types, bypassing the type-specific dispatch characteristic of *ad hoc polymorphism* [82]. To reconcile high-level generality with hardware efficiency, many statically typed languages—including C++ [84], Rust [62], Go [35], MLton [20, 94], and Futhark [41]—leverage compile-time *monomorphization* [57]. The monomorphization process generates a dedicated, specialized version of a generic function for each concrete type instantiation. By statically resolving types at compile-time, monomorphization eliminates the

[1] Functions and data structures defined through parametric polymorphism are referred to as generic functions and generic data types, respectively; these abstractions constitute the foundational building blocks of generic programming [67, 7].

need for heap-allocated indirections and the overhead of dynamic dispatch, fulfilling the zero-cost promise. However, the applicability of monomorphization is not universal. While effective for *predicative* type systems (i.e., rank-1 polymorphism), its complexity escalates significantly within higher-ranked type systems [31, 47, 43].[2] Lutze et al. [57] presented a foundational study on monomorphization that is not only accessible but also generalizes to higher-rank and existential polymorphism. In practice, optimizing compilers [1, 48, 27] rely on interprocedural data-flow analyses to perform *procedure cloning* [25, 26]. They create specialized copies of function bodies for specific type arguments, enabling aggressive optimizations such as inlining [80], constant propagation [17, 95], and dead code elimination [51, 66].[3]

Specialization. Beyond automatic monomorphization, *specialization* represents a sophisticated form of *ad hoc polymorphism*. It permits developers to refine generic abstractions by providing manual, high-priority implementations for specific type instantiations, overriding general-purpose logic with tailored behavior. This practice enables further performance gains by exploiting hardware-specific instructions (e.g., SIMD) or eliminating logic that becomes redundant under specific data characteristics [3]. Specialization is not merely confined to functions; it extends to polymorphic interface types, such as traits in Rust and type classes in Haskell. These constructs enable the definition of behavior that can be specialized based on the types that implement them. However, specializing such interfaces introduces additional complexities, particularly concerning *coherence* [76, 44, 28] and *overlapping instances* [86]. Coherence ensures a unique implementation for each type, thereby preventing ambiguity in method resolution; in Rust, this is enforced through the *orphan rules*.[4] Overlapping instances arise when multiple implementations could apply to the same type, leading to potential conflicts and inconsistencies. Many programming languages have explored various mechanisms to facilitate specialization, with varying degrees of automation and user control. For instance, C++ templates [84] allow for *explicit* specialization of template functions,[5] while the Rust community has introduced an experimental specialization feature [59]. Nevertheless, to date, this feature remains confined to the *nightly* channel [79], as stabilization attempts have stalled due to potential soundness issues and implementation complexities [90].

Limitations. Rust is not unique in confronting challenges when implementing specialization features. The *Project Valhalla*[6] for Java, which aims to introduce value types and generic specialization,[7] has encountered significant hurdles related to backward compatibility and runtime performance, which led to the adoption of *type erasure*. Scala's `@specialized` annotation [69] allows for generating specialized versions of generic classes for primitive types, thereby avoiding boxing overhead. However, the exponential code bloat resulting from multiple type specializations has raised concerns regarding maintainability and compilation times. While Haskell cannot employ template-based specialization [81, 58], its `SPECIALIZE` pragma [74] leverages the *dictionary-passing* implementation of type classes [75] to generate specialized versions of functions for specific type class instances. However, the undecidability of polymorphic recursion [38, 49] and higher-rank types complicates the specialization process, often necessitating manual intervention to guide the compiler [70].

Motivation. Specialization is a potent tool for optimizing performance-critical code sec-

[2] https://okmij.org/ftp/Computation/typeclass.html
[3] For additional information, we refer readers to [8].
[4] https://doc.rust-lang.org/reference/items/implementations.html#orphan-rules
[5] *Partial* template specialization, conversely, aims at specializing class templates based on a subset of their template parameters.
[6] https://openjdk.org/projects/valhalla/
[7] https://mail.openjdk.org/pipermail/valhalla-dev/2014-July/000000.html

tions, particularly in systems programming and high-performance computing domains. Furthermore, it can enhance code clarity and maintainability by encapsulating type-specific logic within specialized implementations, thereby reducing the need for complex conditional logic in generic code. Languages lacking robust specialization mechanisms often compel developers to resort to workarounds, such as manual code duplication or intricate type-level programming, which can lead to code bloat and maintenance challenges. Consequently, there is a pressing need for effective specialization mechanisms that balance performance, usability, and maintainability. For instance, consider the Rust code snippet in Listing 1. The interface type `Trait` is implemented for both **`i32`** and all other types $\forall$`T` where `T` $\neq$ **`i32`**. As shown in Listing 2, Rust's stable toolchain currently rejects this code due to overlapping implementations, a direct consequence of its unimplemented specialization support. Although this specialization pattern can be emulated within the host language (see Listing 1), the implementation complexities and potential soundness issues have historically hindered the development of robust specialization mechanisms in many languages. We contend that, provided the host language supports metaprogramming, specialization can be effectively realized through compile-time code generation, circumventing the need for invasive modifications to the language's type system or compiler infrastructure.

```
trait Trait { fn f(&self); }

impl<T> Trait for T {
    fn f(&self) { ... }
}

impl Trait for i32 {
    fn f(&self) { ... }
}
```

Listing 1 A motivating example.

```
error[E0119]: conflicting implementations
of trait `Trait` for type `i32`
  --> <source>:7:1
   |
3  | impl<T> Trait for T {
   | ------------------- first implementation here
...
7  | impl Trait for i32 {
   | ^^^^^^^^^^^^^^^^^^ conflicting
implementation for `i32`
```

Listing 2 The compilation error of Listing 1.

Proposal. In this manuscript, we introduce *meta-monomorphizing specializations*, a novel approach that leverages compile-time metaprogramming to generate specialized versions of polymorphic functions and interface types, building upon the principles of monomorphization. Our central idea is to employ monomorphization as a foundation for specialization, where the compiler automatically generates specialized implementations based on user-defined directives. Metaprogramming has evolved into a fundamental paradigm in modern language design, as highlighted by Lilis and Savidis [56].

Our approach operates specifically with the metaprogramming facilities (e.g., macros and code generation) available in the host language [56], allowing developers to define specialization logic in a high-level, declarative manner. Crucially, we aim to preserve the compilation pipeline of the host language, ensuring compatibility while reusing existing type-checking passes and optimization strategies. The snippet in Listing 3 illustrates how our approach resolves the specialization issue through the `#[when(T = i32)]` attribute—a procedural macro (cf. §2) that generates the specialized implementation for the ground type **`i32`**. While the proposed approach is fundamentally language-agnostic, we have selected the Rust programming language

```
trait Trait { fn f(&self); }
impl<T> Trait for T {
    fn f(&self) { ... }
}
#[when(T = i32)]
impl<T> Trait for T {
    fn f(&self) { ... }
}
```

Listing 3 Specialization solved through meta-monomorphization.

as the target for our study, given its strong emphasis on performance, safety, and modern metaprogramming features.

Non-goals. Our focus is exclusively on enabling meta-monomorphic specializations. Consequently, we assume fully type-annotated programs as input, leaving type inference as a potential preprocessing step. While monomorphization inherently involves whole-program analysis and code duplication, optimizing the resulting transformation time, binary size, or redundancy is outside the scope of this work; however, modern optimizing compilers perform dead or unreachable code elimination. Improving the precision of our analysis, however, remains an interesting and important direction for future work. Finally, while our approach targets combinations of:

- first-order programs with equality bound (§3.1);
- predicate polymorphism with trait bounds (§3.2);
- polymorphic $\sum$- and $\prod$-type constructors (§3.3);
- lifetime polymorphism with reference types (§3.4); and
- higher-ranked polymorphism with higher-order functions (§3.5)

It does not currently support all program classes, such as recursive polymorphic functions [38, 11, 37, 78] or existential types [65, 54] (cf. §3.6). Instead, we provide a practical specialization framework that leverages existing metaprogramming without requiring compiler modifications.

Contributions. The contributions of this work are threefold:

- We introduce the concept of *meta-monomorphizing specializations*, detailing its design principles, formalization, and implementation strategies.
- We present a metaprogramming framework, designed, developed, and extensively tested, that facilitates the definition and generation of specialized implementations.
- We conduct an engineering study on higher-ranked codebases to assess the practical implications and benefits of our method.

Structure. The rest of this paper is organized as follows. §2 introduces Rust's type system and macro facilities. §3 details our approach through progressive examples. §4 presents an empirical study on public Rust codebases. §5 discusses potential threats to the validity of our results. §6 surveys related work, and §7 concludes.

2 The Rust Programming Language

Rust [62] is a systems programming language designed around the principles of safety, speed, and concurrency. It guarantees memory safety without garbage collection, meaning that pure Rust programs are provably free from null pointer dereferences and unsynchronized race conditions at compile time [45].

Ownership and Borrowing. Rust's ownership system, integrated into its type system, is formally grounded in *linear logic* [33, 34] and *linear types* [92, 68], enforcing that each value has a single *owner* (a variable binding) at any given time [22, 12]. When the owner goes out of scope, the associated memory is automatically deallocated, enabling user-defined destructors and supporting the *resource acquisition is initialization* (RAII) pattern [83]. Ownership can be transferred (*moved*) or temporarily shared (*borrowed*) through references. Rust imposes a strict borrowing discipline: at any given time, a piece of data can have either multiple *immutable* borrows or a single *mutable* borrow, but not both simultaneously. This invariant is enforced through strict aliasing rules over references. Mutable references (`&mut T`) guarantee exclusive access to the underlying data, while immutable references (`&T`) permit shared access.

These constraints, enforced by the compiler's *borrow checker*, guarantee memory safety and prevent dangling pointers by ensuring that reference lifetimes never outlive their owners. To support low-level operations, Rust provides **`unsafe`** blocks, wherein the compiler's safety guarantees are suspended, and the burden of avoiding undefined behavior (UB) falls upon the programmer. Outside these designated blocks, Rust enforces strict safety, making UB impossible in safe code.

Type System. In addition to primitive types such as **`i32`** and **`bool`**, Rust supports both $\sum$- and $\prod$-types, realized as algebraic data types (ADTs). The former are represented by the **`enum`** keyword, and the latter by the **`struct`** keyword. These composite types can be made polymorphic through generic type parameters [46], which allow for the definition of types and functions that operate over a variety of data types while maintaining compile-time type safety. Rust also features robust pattern matching, which facilitates the concise and expressive handling of aggregate data types by deconstructing them and matching against their internal structure. Pattern matching is tightly integrated with Rust's type system, enabling sophisticated error handling and control flow, as exemplified by standard library types such as `Option<T>` and `Result<T, E>`. The type system is further enriched by interface types, declared using the **`trait`** keyword. Traits define shared behavior that concrete types can implement, supporting both static and dynamic dispatch. This mechanism enables abstraction over operations and facilitates code reuse while preserving strong type safety. A trait can be utilized in three primary forms: as a *bounded type parameter* via `<T: Trait>`, as an *existential type* via `impl Trait`, or as a *trait object* via `dyn Trait`. However, the expressiveness of the trait system leads to undecidable type checking in the general case. Existential polymorphism, through `impl Trait`, allows functions to return types that implement a specific trait while keeping the concrete type opaque to the caller, thereby enhancing abstraction and encapsulation. Another form of existential polymorphism is realized through `&dyn Trait`, which enables function parameters to accept references to any object that implements a trait, without requiring the concrete type to be known at compile time. Lifetimes are another cornerstone of Rust's type system. The temporal validity of a reference is governed by its lifetime. While often elided in source code for brevity, the full form of a reference type—`&'a T` or `&'a mut T`—explicitly annotates the duration for which the borrow remains valid. A precise understanding of these lifetimes is crucial for the type-checking process, as they allow the compiler to guarantee that no reference outlives the data it points to. This property is particularly evident in the following example.

```rust
fn bar() {
    'a: {
        let res;
        'b: {
            let x = 7;
            res = &/*'b */x;
        }
        println!("{res}");
    }
}
```

```
error[E0597]: `x` does not live long enough
 --> <source>:6:19
  |
5 | let x = 7;
  |     - binding `x` declared here
6 | res = &/*'b */x;
  |       ^^^^^^^^^ borrowed value does
                    not live long enough
7 | }
  | - `x` dropped here while still borrowed
8 | println!("{res}");
  |          --- borrow later used here
error: aborting due to 1 previous error
```

The compiler must raise an error because the reference `res` is assigned the address of `x`, which is declared in the inner scope `'b` and goes out of scope (and is thus dropped) at the end of that block. When `res` is used in the `'a` scope, it points to a value that no longer exists, leading to a dangling reference.

Subtyping. Rust supports subtyping. In particular, lifetimes form a subtyping hierarchy based on their scopes. We provide a parallel between Rust's lifetime subtyping and natural deduction expressed through Gentzen-style subtyping deduction rules [73]. The notation $T <: U$ indicates that type T is a subtype of type U. A lifetime `'a` is a subtype of another lifetime `'b` if the scope of `'a` is contained within the scope of `'b`—i.e., `'a` *outlives* `'b`, denoted as `'a: 'b` (cf. $\boxed{\text{L-Sub}}$). The variance difference between shared and mutable references is at the heart of memory safety in Rust. Shared references are *covariant* over both their lifetime and the type they point to (cf. $\boxed{\text{Ref-Cov}}$). Mutable References, on the other hand, are *invariant* over the type they point to, while remaining covariant over their lifetime (cf. $\boxed{\text{RefMut-Inv}}$). Function pointers also exhibit variance properties. They are *contravariant* in their argument types and *covariant* in their return types (cf. $\boxed{\text{Fn-Sub}}$). Intuitively, this means that a function that accepts more general arguments and returns more specific results can be used wherever a function with more specific arguments and more general results is expected.

Rust supports smart pointers, such as `Box<T>` (heap allocation), and containers, such as `Vec<T>` (dynamic arrays). Both *own* their data and do not allow aliasing (cf. $\boxed{\text{BoxVec-Sub}}$). The interior mutability provided by types like `Cell<T>` and `UnsafeCell<T>` allows for mutation through shared references. They are invariant over the type they contain to prevent unsoundness (cf. $\boxed{\text{CellUns-Inv}}$). Finally, raw pointers (`*const T` and `*mut T`) are similar to C/C++ pointers and do not enforce any ownership or borrowing rules. They follow the same variance rules as references (cf. $\boxed{\text{PtrConst-Cov}}$ and $\boxed{\text{PtrMut-Inv}}$).

$$\frac{\texttt{'a: 'b}}{\texttt{'a} <: \texttt{'b}}\ \boxed{\text{L-Sub}}$$

$$\frac{\texttt{'a} <: \texttt{'b} \quad T <: U}{\texttt{\&'a}\ T <: \texttt{\&'b}\ U}\ \boxed{\text{Ref-Cov}}$$

$$\frac{\texttt{'a} <: \texttt{'b} \quad T = U}{\texttt{\&'a mut}\ T <: \texttt{\&'b mut}\ U}\ \boxed{\text{RefMut-Inv}}$$

$$\frac{T_1 <: T_2 \quad U_1 <: U_2}{\texttt{fn(}T_2\texttt{) ->}\ U_1 <: \texttt{fn(}T_1\texttt{) ->}U_2}\ \boxed{\text{Fn-Sub}}$$

$$\frac{T <: U \quad \mathcal{F} \in \{\texttt{Box}, \texttt{Vec}\}}{\mathcal{F}\texttt{<}T\texttt{>} <: \mathcal{F}\texttt{<}U\texttt{>}}\ \boxed{\text{BoxVec-Sub}}$$

$$\frac{T = U \quad \mathcal{F} \in \{\texttt{Cell}, \texttt{UnsafeCell}\}}{\mathcal{F}\texttt{<}T\texttt{>} <: \mathcal{F}\texttt{<}U\texttt{>}}\ \boxed{\text{CellUns-Inv}}$$

$$\frac{T <: U}{\texttt{*const}\ T <: \texttt{*const}\ U}\ \boxed{\text{PtrConst-Cov}}$$

$$\frac{T = U}{\texttt{*mut}\ T <: \texttt{*mut}\ U}\ \boxed{\text{PtrMut-Inv}}$$

Higher-Rank Polymorphism. Rust does not merely support rank-1 polymorphism through generics; it also embraces higher-rank polymorphism via *higher-ranked trait bounds* (HRTBs) [50, 87]. HRTBs allow functions to be generic over lifetimes that are not known until the function is called, enabling more flexible and reusable abstractions. This is achieved through the `for<'a>` syntax, which specifies that a type or function is valid for all choices of lifetime `'a` (i.e., ∀`'a`). As an example, consider the following `apply` function, which takes a reference to an `i32` and a closure `f` that can accept a reference with any lifetime:

```
fn apply<F, T>(p: &i32, f: F) -> T where F: for<'a> Fn(&'a i32) -> T { f(p) }
```

This function can be called with closures that accept references with different lifetimes, demonstrating the power of higher-rank polymorphism in Rust.

Macro System. Rust's macro system allows for metaprogramming by enabling code generation and transformation at compile time. There are two main types of macros in Rust: declarative macros (using `macro_rules!`) and procedural macros. The former operate through pattern matching over token trees, allowing developers to define reusable code snippets that can be invoked with different arguments. While reminiscent of Lisp macros in spirit [63], Rust's declarative macros enforce *hygiene* [52, 39, 24]: macro-generated identifiers cannot inadvertently capture variables or introduce unintended bindings. The latter, procedural macros, operate on the abstract syntax tree (AST) of the code, allowing for more complex transformations and code generation. They come in three forms: *function-like macros*, *custom derive macros*, and *attribute-like macros*. Function-like macros resemble functions but operate on token streams, enabling developers to create domain-specific languages [88] or perform complex code manipulations [21]. Custom derive macros are widely used to automatically implement traits for user-defined types, such as `Clone` or `Debug`. Attribute-like macros allow for annotating items with custom attributes that can modify their behavior or generate additional code; the `#[when(...)]` attribute macro highlighted in §1 is an example of such a use.

3 Meta-Monomorphizing Specializations by Examples

This section elucidates our approach through a progressive sequence of examples, each designed to reveal a distinct layer of the meta-monomorphization process. Each stage builds upon the preceding one, with all established properties and assumptions persisting throughout. Our objective is to provide a high-level yet compiler-accurate exposition of the core mechanics. A discussion of limitations is deferred to §3.6. In the following, *the compiler* refers in particular to the macro expansion phase of the Rust compiler.

3.1 First-Order Programs with Equality Bounds

We begin by considering the program in Listing 4. This example extends the motivating scenario from §1 by parameterizing `trait Trait` over a type `T` and augmenting its method `f` to accept an argument of this type.

In this configuration, two distinct implementations of `Trait<T>` are provided for the type `ZST` (a zero-sized type). The first is a specialized variant, constrained by a *formal* equality specialization bound requiring `T = i32`. The second is a generic fallback for all other types. Within the `main` function, the method `f` is invoked

```
struct ZST;
trait Trait<T> { fn f(&self, a: T); }
#[when(T = i32)]
impl<T> Trait<T> for ZST { fn f(&self, a: T) {} }
impl<T> Trait<T> for ZST { fn f(&self, a: T) {} }
fn main() { let s = ZST;
    spec! { s.f("s"); ZST; [ _ ] }
    spec! { s.f( 7 ); ZST; [i32] } }
```

Listing 4 A first-order program with trait specializations.

twice on a `ZST` instance, first with a `&str` argument and subsequently with an `i32`, necessitating dispatch to the appropriate implementation in each case. Our function-like macro, `spec!`, serves as a crucial marker, enabling our meta-monomorphization procedure to identify *specialized call sites*. This macro accepts three arguments: the method call expression, the *receiver type* (e.g., `ZST`), and a list of *actual* specialization parameter bounds. These bounds consist of either concrete types (e.g., `[i32]`) or a wildcard `[_]` to signify the absence of specialization.[8] Henceforth, we will use the abbreviation SBs for *specialization bounds*, rendered in boldface (e.g., $\mathbf{B_1}$) to distinguish them from type parameters (e.g., T_1). For clarity of presentation, we assume all type parameters are uniquely named.

The meta-monomorphization of specializations in first-order programs proceeds according to the following sequence of compiler transformations:

1. **Meta-Monomorphizing Traits.** For each `#[when(...)]` specialization attribute, the compiler synthesizes a *distinctly named* meta-monomorphized trait definition. This new trait is a specialized version of the original, tailored to a specific formal SB. Let $\mathcal{T}$ be a trait with n type parameters $T_1, \ldots, T_n$, and let $\mathbf{B_1}$ be a ground type serving as a formal SB for the type parameter T_1. For every specialization implementation of the form:

   ```
   #[when(T₁ = B₁)]
   impl<T₁,...,Tₙ> 𝒯<T₁,...,Tₙ> for 𝒮 {
       fn f(&self, a1: T₁, ..., an: Tₙ) { ... } }
   ```

   the compiler generates a meta-monomorphized trait definition:

   ```
   trait 𝒯^[B₁]<T₂,...,Tₙ> {
       fn f(&self, a1: B₁, a2: T₂, ..., an: Tₙ); }
   ```

   Here, the new trait name $\mathcal{T}^{[\mathbf{B_1}]}$ indicates that the first formal parameter is now bound to the ground type $\mathbf{B_1}$, while the remaining parameters $T_2, \ldots, T_n$ are preserved as generic. This procedure is applied systematically to all *associated items* (e.g., methods, type aliases) within the trait. The resulting set of all such generated traits, augmented with the original *default* trait, is denoted as $\mathrm{M} = \{\mathcal{T}^{[\mathbf{B_1}]}\} \cup \{\mathcal{T}\}$.[9]

2. **Specialization Extraction.** For each specialization, the compiler extracts its body to generate a corresponding implementation of the newly created meta-monomorphized trait. Assuming $\mathcal{T}^{[\mathbf{B_1}]} \in \mathrm{M}$, the original implementation is transformed into:

   ```
   impl<T₂,...,Tₙ> 𝒯^[B₁]<T₂,...,Tₙ> for 𝒮 {
       fn f(&self, a1: B₁, a2: T₂, ..., an: Tₙ) { ... } }
   ```

   Crucially, the type parameter T_1 is replaced by the concrete type $\mathbf{B_1}$ in the method signature. This process is repeated for all associated items. The set of all generated implementations for a concrete type $\mathcal{S}$ is denoted $\mathrm{I} = \{(\mathcal{S}, \mathcal{T}^{[\mathbf{B_1}]})\}$. If multiple specializations declare overlapping SBs, the system generates distinct meta-monomorphized traits and implementations for each, deferring overlap resolution to the subsequent stage.

3. **Overlapping Instances Checking.** To guarantee deterministic dispatch, as established in §1, specialization implementations must not have overlapping formal SBs. This property is enforced through a static analysis pass that checks for overlapping instances, a problem

[8] While these bounds could be inferred via compile-time reflection, such mechanisms are orthogonal to the core contribution and thus beyond the scope of this paper (cf. Non-goals in §1).

[9] Generating traits for all declared specializations might appear suboptimal if some are unused. However, the Rust compiler's dead-code elimination pass [51] effectively removes unreferenced trait definitions. An alternative, demand-driven strategy that generates traits only for utilized specializations is a viable area for future work.

known to be undecidable in its general form [5]. For any pair of implementations in I for the same type $\mathcal{S}$ but different meta-monomorphized traits ($\mathcal{T}^{[\mathbf{B_1}]}$ and $\mathcal{T}^{[\mathbf{C_1}]}$), the compiler checks for SB overlap. Formally, it determines whether a *unifying* substitution σ exists such that $\sigma(\mathbf{B_1}) \equiv \sigma(\mathbf{C_1})$. A unifier $\sigma = \{T \mapsto U\}$ exists if applying it to both $\mathbf{B_1}$ and $\mathbf{C_1}$ yields an identical type. Our system permits overlaps only when one specialization is strictly more specific than the other (further details are provided in §3.7). This check not only flags ambiguous SBs with a compiler error but also ensures that actual SBs at any call site will match at most one specialization.

4. **Specialization Bounds Coherence Checking.** To ensure coherence at each call site marked with the `spec!` macro, the compiler must select the appropriate specialization. This is achieved by matching the *actual* SBs from the call site against the *formal* SBs of all implementations in I. Given a call site of the form:

   $$\texttt{spec! \{ s.f(}a_1, a_2, \ldots, a_n\texttt{); } \mathcal{S}\texttt{; [}\mathbf{B_1}\texttt{]; \}}$$

   where `s:` $\mathcal{S}$ is the receiver, $a_1, \ldots, a_n$ are the arguments, and $[\mathbf{B_1}]$ is the actual SB, the compiler searches for a unique implementation $(\mathcal{S}, \mathcal{T}^{[\mathbf{B_1}]}) \in$ I whose formal SBs are equivalent to the actual SBs. The preceding overlap check guarantees that no more than one such implementation can exist for a well-formed program. In first-order programs, this matching is straightforward, as both formal and actual SBs are ground types. If a matching specialization $\mathcal{T}^{\mathbf{B_1}}$ is found, it is selected for the final transformation; otherwise, the call site defaults to the non-specialized trait $\mathcal{T}$.

5. **Call Site Specialization.** The final stage is to rewrite each specialized call site to invoke the method from the uniquely matched meta-monomorphized trait. Given the call site above and the matched implementation $(\mathcal{S}, \mathcal{T}^{[\mathbf{B_1}]}) \in$ I, the compiler rewrites the call as:

   $$\texttt{<}\mathcal{S}\texttt{ as } \mathcal{T}^{[\mathbf{B_1}]}\texttt{>::f(\&s, } a_1, a_2, \ldots, a_n\texttt{);}$$

   This rewrite employs Rust's *fully qualified syntax* to explicitly name the receiver type $\mathcal{S}$ and the meta-monomorphized trait $\mathcal{T}^{[\mathbf{B_1}]}$, thereby ensuring dispatch to the correct specialized implementation. Non-specialized call sites are similarly rewritten to invoke the default trait $\mathcal{T}$: $\texttt{<}\mathcal{S}\texttt{ as } \mathcal{T}\texttt{>::f(\&s, } a_1, \ldots, a_n\texttt{);}$

Following these transformations, the program is internally rewritten during the compiler's macro expansion phase to employ standard, non-specialized trait implementations. This transformed program is then lowered through the conventional compilation pipeline, which includes type-checking the HIR (High-level Intermediate Representation), borrow-checking the MIR (Mid-level Intermediate Representation), monomorphizing generics, and finally code generation (e.g., to LLVM IR [53]). In the case of the program in Listing 4, after applying our meta-monomorphization procedure, the transformed program would look as follows:

```
struct ZST;
trait Trait<T> { fn f(&self, a: T); }
trait Trait_i32 { fn f(&self, a: i32); }
impl<T> Trait<T> for ZST { fn f(&self, a: T) {} }
impl Trait_i32 for ZST { fn f(&self, a: i32) {} }
```

```
fn main() {
    let s = ZST;
    <ZST as Trait<&str>>::f(&s, "s");
    <ZST as Trait_i32>::f(&s, 42); }
```

3.2 Predicate Polymorphism with Trait Bounds

To illustrate predicate polymorphism, we adapt the program from Listing 4. The formal SB is changed from a simple equality `T = i32` to a compound *predicate*, `any(T = i32, T:`

```
struct ZST;
trait Trait<T> { fn f(&self, a: T); }
#[when(any(T = i32, T: Clone))]
impl<T> Trait<T> for ZST { fn f(&self, a: T) {} }
impl<T> Trait<T> for ZST { fn f(&self, a: T) {} }
```

```
fn main() { let s = ZST;
    spec! { s.f("s"); ZST; [ _ ] }
    spec! { s.f( vec![1] );
    ZST; [Vec<i32>];
    Vec<i32>: Clone } }
```

1 Predicate polymorphism and trait specializations.

`Clone`), which is satisfied either by the ground type `i32` or by any type implementing the trait `Clone` (cf. Listing 1). Consequently, the `spec!` macro is extended to accept the trait bounds required to satisfy the actual SBs at a given call site (e.g., `Vec<i32>: Clone`). The dispatch for the first call to `f` remains unaltered, whereas the second call now resolves to the specialized implementation, since `Vec<i32>` implements `Clone`.

Meta-monomorphizing specializations under predicate polymorphism proceeds analogously to the first-order case, with several key adaptations:

1. Let $\hat{P}(P_1, \ldots, P_x)$ be a recursive predicate formula where each clause P_i is either (i) an equality SB T_i `=` $\mathbf{B_i}$ or a trait SB T_i`:` $\boldsymbol{\mathcal{T}_i}$, or (ii) a *nested predicate* from the set {`any`, `all`, `not`} over such atoms. Without loss of generality, we assume $\hat{P}$ has been *canonicalized* into Disjunctive Normal Form (DNF)[10]—i.e., it is expressed as `any(`$P_1, \ldots, P_x$`)`. For each specialization implementation governed by such a predicate:

   ```
   #[when(P̂(P_1, ..., P_x))]
   impl<T_1, ..., T_n> 𝒯<T_1, ..., T_n> for 𝒮 {
       fn f(&self, a1: T_1, ..., an: T_n) { ... } }
   ```

   the compiler generates, for each disjunct P_i, a distinctly named meta-monomorphized trait definition:

   ```
   trait 𝒯^[P_i]<T_{k+1}, ..., T_{k+1+l}, T_{k+l+2}, ..., T_n> {
       fn f(&self, a1: B_1, ..., ak: B_k, ❶ // Equality Bounded in 𝒯
            akp1: T_{k+1}, ..., akp1pl: T_{k+1+l}, ❷ // Trait Bounded in 𝒯
            akplp2: T_{k+l+2}, ..., an: T_n); } ❸ // Generic in 𝒯
   ```

   where the disjunct P_i is composed of k formal equality SBs T_1 `=` $\mathbf{B_1}, \ldots, T_k$ `=` $\mathbf{B_k}$ (cf. ❶), l formal trait SBs T_{k+1}`:` $\boldsymbol{\mathcal{T}_{k+1}}, \ldots, T_{k+1+l}$`:` $\boldsymbol{\mathcal{T}_{k+1+l}}$ (cf. ❷), and the remaining generic type parameters $T_{k+l+2}, \ldots, T_n$ (cf. ❸). The trait SBs are preserved as generic type parameters in the synthesized trait. The set $\hat{\mathrm{M}}$ thus extends M to include meta-monomorphized traits for each predicate disjunct P_i.

2. For each specialization, the compiler generates a corresponding implementation for every associated meta-monomorphized trait in $\hat{\mathrm{M}}$. The previously generic trait SB is now concretized by substituting the corresponding type parameters with their bounds within the `impl` block.

   ```
   impl<T_{k+1}: 𝒯_{k+1}, ..., T_{k+1+l}: 𝒯_{k+1+l}, ❷
           T_{k+l+2}, ..., T_n ❸> 𝒯^[P_i]<T_{k+1}, ..., T_n> for 𝒮 {
       fn f(&self, a1: B_1, ..., ak: B_k, ❶
            akp1: T_{k+1}, ..., an: T_n ❷ ❸) { ... } }
   ```

   It is possible for the same type parameter to appear in both equality and trait SBs

[10] The canonicalization process for predicate formulas is detailed in §3.7.

within a disjunct P_i (e.g., `all(T = Vec<i32>, T: Clone)`). In such cases, if the equality SB implies the trait SB (since `Vec<i32>` implements `Clone`), the trait SB can be safely elided from the parameter list. The set $\hat{\mathrm{I}} = \mathrm{I} \cup \{(\mathcal{S}, \mathcal{T}^{[P_i]})\}$ extends I with these newly generated implementations.

3. In contrast to first-order programs, predicate-based specializations introduce multiple sources of potential overlap. Formal SBs within a single disjunct P_i may conflict, as can different disjuncts P_i and P_j of the same predicate $\hat{P}$. The compiler first checks for intra-disjunct consistency, ensuring no two atoms for the same type parameter T are contradictory (e.g., `T = i32` and `T: Debug` are compatible, whereas `T = i32` and `T = bool` are not). Second, for every pair of implementations in $\hat{\mathrm{I}}$ for the same type $\mathcal{S}$ but with different meta-monomorphized traits $\mathcal{T}^{[P_i]}$ and $\mathcal{T}^{[Q_j]}$, it checks for a unifying substitution σ where $\sigma(P_i) \equiv \sigma(Q_j)$. Finally, as in the first-order case, inter-trait overlaps are checked between all pairs in $\hat{\mathrm{M}}$ by comparing their respective formal SBs.

4. Coherence checking is extended to accommodate predicate SBs. The `spec!` macro becomes *variadic*, accepting an arbitrary number of trait bounds as actual SBs. Given a call site of the form:

$$\texttt{spec! \{ s.f(}a_1,\ldots,a_k,a_{k+1},\ldots,a_{k+1+l},a_{k+l+2},\ldots,a_n\texttt{);}$$
$$\mathcal{S}\texttt{; [}\mathbf{B_1},\ldots,\mathbf{B_k}\texttt{]; } T_{k+1}\texttt{: } \boldsymbol{\mathcal{T}_{k+1}}\texttt{, ..., } T_{k+1+l}\texttt{: } \boldsymbol{\mathcal{T}_{k+1+l}}\texttt{ \}}$$

the compiler must find a unique implementation $(\mathcal{S}, \mathcal{T}^{[P_i]}) \in \hat{\mathrm{I}}$ whose formal SBs P_i match the actual SBs. The matching logic must now handle ground type equivalence, trait bound satisfaction, and predicate unification. Actual equality SBs may themselves be generic (e.g., `Vec<U>`). In such cases, the matching procedure must ensure that type parameters like U are instantiated consistently across all SBs at the call site. A key distinction from the first-order case is that multiple disjuncts P_i from the same predicate $\hat{P}$ may match the actual SBs. The compiler must therefore select the *most specific* implementation among the candidates (cf. §3.7).

5. This step remains functionally identical to the first-order case, with the distinction that the rewritten call site may now invoke a method from a meta-monomorphized trait $\mathcal{T}^{[P_i]}$ corresponding to a predicate formula P_i.

```
use std::marker::PhantomData;
struct ZST<U>(PhantomData<U>);
trait Trait<T> { fn f(&self, a: T); }
#[when(all(any(T = i32, T: Clone), U = bool))]
impl<T,U> Trait<T> for ZST<U> {fn f(&self, a:T){}}
impl<T,U> Trait<T> for ZST<U> {fn f(&self, a:T){}}
```

```
fn main() {
  let s = ZST::<bool>(PhantomData);
  let t = ZST::<u8>(PhantomData);
    spec!{s.f(7); ZST<bool>; [i32]}
    spec!{t.f(7); ZST<u8>; [i32]}}
```

2 A program with polymorphic type constructors and nested predicate specializations. The `PhantomData` is used to indicate that `ZST` is generic over `U` without actually storing a value of type `U`.

3.3 Polymorphic $\sum$- and $\prod$-type Constructors

As noted in §2, Rust supports polymorphic $\sum$- and $\prod$-type constructors. To demonstrate how our approach accommodates such constructors, we modify the program in Listing 1. A type parameter `U` is added to the `ZST` struct, and the formal SB is refined to a nested predicate: `all(any(T = i32, T: Clone), U = bool)`. This predicate matches the previous conditions on `T` while also requiring that `U` be bound to `bool` (cf. Listing 2). The `spec!` macro must now receive the receiver type with its full type arguments (e.g., `ZST<bool>`).

Consequently, only the first call to `f` dispatches to the specialized implementation, as the receiver type `ZST<u8>` in the second call fails to satisfy the formal SB `U = bool`.

The meta-monomorphization procedure requires further adaptation to handle type parameters from the implementing type $\mathcal{S}$:[11]

1. When synthesizing meta-monomorphized traits, type parameters from the implementing type are incorporated *only if* they also appear in the trait instantiation. Let $\tilde{P}(P_1, \ldots, P_x)$ be a DNF predicate formula that may now reference type parameters from the implementing type $\mathcal{S}$. Consider a specialization of the form:

$$
\begin{array}{l}
\texttt{\#[when(}\tilde{P}(P_1,\ldots,P_x)\texttt{)]}\\
\texttt{impl<}\underbrace{T_1,\ldots,T_m}_{❶❷❸}\texttt{, }\underbrace{T_{m+1},\ldots,T_{m+1+o}}_{❹❺❻}\texttt{, }\underbrace{T_{m+o+2},\ldots,T_n}_{❼❽❾}\texttt{> }\mathcal{T}\texttt{<}\underbrace{T_1,\ldots,T_{m+1+o}}_{❶,\ldots,❻}\texttt{>}\\
\quad\texttt{for }\mathcal{S}\texttt{<}\underbrace{T_{m+1},\ldots,T_n}_{❹,\ldots,❾}\texttt{> \{}\\
\qquad\texttt{fn f(\&self, a1: }T_1\texttt{, ..., a1po: }T_{m+1+o}\texttt{) \{ ... \} \}}
\end{array}
$$

   where ❹, ❺, and ❻ denote type parameters shared between the trait and the implementing type $\mathcal{S}$. For each disjunct P_i, the compiler generates a meta-monomorphized trait:

$$
\begin{array}{l}
\texttt{trait }\mathcal{T}^{[P_i]}\texttt{<}\underbrace{T_{k+1},\ldots,T_m}_{❷❸}\texttt{, }\underbrace{T_{m+r+2},\ldots,T_{m+1+o}}_{❺❻}\texttt{> \{}\\
\quad\texttt{fn f(\&self, a1: }\mathbf{B_1}\texttt{, ..., ak: }\mathbf{B_k}\texttt{, ❶ // Equality Bounded (EB) in }\mathcal{T}\\
\qquad\texttt{akp1: }T_{k+1}\texttt{, ..., akp1pl: }T_{k+1+l}\texttt{, ❷ // Trait Bounded (TB) in }\mathcal{T}\\
\qquad\texttt{akplp2: }T_{k+l+2}\texttt{, ..., am: }T_m\texttt{, ❸ // Generic (Gen) in }\mathcal{T}\\
\qquad\texttt{amp1: }\mathbf{B_{m+1}}\texttt{, ..., amp1pr: }\mathbf{B_{m+1+r}}\texttt{, ❹ // EB in }\mathcal{S}\texttt{ \& }\mathcal{T}\\
\qquad\texttt{amprp2: }T_{m+r+2}\texttt{, ..., amprp2ps: }T_{m+r+2+s}\texttt{, ❺ // TB in }\mathcal{S}\texttt{ \& }\mathcal{T}\\
\qquad\texttt{amprpsp3: }T_{m+r+s+3}\texttt{, ..., am1po: }T_{m+1+o}\texttt{); \} ❻ // Gen in }\mathcal{S}\texttt{ \& }\mathcal{T}
\end{array}
$$

   where, in addition to the categories ❶, ❷, and ❸, we now have: r formal equality SBs (cf. ❹), s formal trait SBs (cf. ❺), and the remaining generic type parameters (cf. ❻) that originate from $\mathcal{S}$ and also appear in the trait instantiation. Type parameters exclusive to $\mathcal{S}$ are handled in the next step. The set $\tilde{\mathbb{M}} = \hat{\mathbb{M}} \cup \{\mathcal{T}^{[P_i]}\}$ extends $\hat{\mathbb{M}}$ with these new traits.
2. To generate implementations, we must now also account for type parameters belonging solely to the implementing type $\mathcal{S}$. Let the predicate P_i of a trait $\mathcal{T}^{[P_i]} \in \tilde{\mathbb{M}}$ contain:
   - ❼ t formal equality SBs T_{m+o+2} `=` $\mathbf{B_{m+o+2}}, \ldots, T_{m+o+2+t}$ `=` $\mathbf{B_{m+o+2+t}}$,
   - ❽ u formal trait SBs $T_{m+o+t+3}$`:` $\boldsymbol{\mathcal{T}_{m+o+t+3}}, \ldots, T_{m+o+t+3+u}$`:` $\boldsymbol{\mathcal{T}_{m+o+t+3+u}}$, and
   - ❾ remaining generic type parameters $T_{m+o+t+u+4}, \ldots, T_n$

   that are part of $\mathcal{S}$ but **do not** appear in the trait instantiation. A corresponding implementation is generated for each meta-monomorphized trait in $\tilde{\mathbb{M}}$ as follows:

$$
\begin{array}{l}
\texttt{impl<}T_{k+1}\texttt{: }\boldsymbol{\mathcal{T}_{k+1}}\texttt{, ..., }T_{k+1+l}\texttt{: }\boldsymbol{\mathcal{T}_{k+1+l}}\texttt{, ❷ // Trait Bounded in }\mathcal{T}\\
\qquad T_{k+l+2}\texttt{, ..., }T_m\texttt{, ❸ // Generic in }\mathcal{T}\\
\qquad T_{m+r+2}\texttt{: }\boldsymbol{\mathcal{T}_{m+r+2}}\texttt{, ..., }T_{m+r+2+s}\texttt{: }\boldsymbol{\mathcal{T}_{m+r+2+s}}\texttt{, ❺ // TB in }\mathcal{S}\texttt{ \& }\mathcal{T}\\
\qquad T_{m+r+s+3}\texttt{, ..., }T_{m+1+o}\texttt{ ❻ // Gen in }\mathcal{S}\texttt{ \& }\mathcal{T}\\
\qquad\underbrace{T_{m+o+t+3},\ldots,T_n}_{❽❾}\texttt{> }\mathcal{T}^{[P_i]}\texttt{<}\underbrace{T_{k+1},\ldots,T_m}_{❷❸}\texttt{, }\underbrace{T_{m+r+2},\ldots,T_{m+1+o}}_{❺❻}\texttt{>}\\
\quad\texttt{for }\mathcal{S}^{[P_i]}\texttt{<}\underbrace{\mathbf{B_{m+1}},\ldots,\mathbf{B_{m+1+r}}}_{❹}\texttt{, }\underbrace{T_{m+r+2},\ldots,T_{m+1+o}}_{❺❻}\texttt{,}
\end{array}
$$

[11] The procedure is identical for both $\sum$- and $\prod$-type constructors; hence, we do not distinguish between them.

$$\underbrace{\mathbf{B_{m+o+2}}\texttt{,}\ldots\texttt{,}\mathbf{B_{m+o+2+t}}}_{❼}\texttt{,}\ \underbrace{T_{m+o+t+3}\texttt{,}\ldots\texttt{,}T_n}_{❽\ ❾}\texttt{> \{}$$

$$\begin{array}{l}
\texttt{fn f(\&self, a1: }\mathbf{B_1}\texttt{, ..., ak: }\mathbf{B_k}\texttt{,}\ ❶\\
\quad\texttt{akp1: }T_{k+1}\texttt{, ..., am: }T_m\texttt{,}\ ❷\ ❸\\
\quad\texttt{amp1: }\mathbf{B_{m+1}}\texttt{, ..., amp1pr: }\mathbf{B_{m+1+r}}\texttt{,}\ ❹\\
\quad\texttt{amprp2: }T_{m+r+2}\texttt{, ..., amp1po: }T_{m+1+o}\ ❺\ ❻\texttt{) \{ ... \} \}}
\end{array}$$

Depending on the structure of $\tilde{P}$, multiple implementations may share the same equality SBs from $\mathcal{S}^{[P_i]}$ (cf. ❼) while differing in trait SBs or generic parameters (cf. ❺, ❻). For example, in Listing 2, specializations for `all(T = i32, U = bool)` and `all(T: Clone, U = bool)` share the equality SB `U = bool` but differ in the SB for `T`. These combinatorial possibilities introduce complexity into the subsequent coherence checks. The set $\tilde{\mathrm{I}} = \hat{\mathrm{I}} \cup \{(\mathcal{S}^{[P_i]}, \mathcal{T}^{[P_i]})\}$ extends $\hat{\mathrm{I}}$ with all such generated implementations.

3. The overlap checking procedure is extended to reason about type parameters from the implementing type $\mathcal{S}^{[P_i]}$. For each pair of implementations in $\tilde{\mathrm{I}}$, the process is twofold:
   - With the implementing types $\mathcal{S}^{[P_i]}$ and $\mathcal{S}^{[Q_j]}$ fixed, let $\delta = P_i \overset{\mathcal{S}}{\cap} Q_j \neq \emptyset$ be the set of common SBs with respect to parameters from $\mathcal{S}$. We check if a unifying substitution σ exists such that $\sigma(P_i) \equiv \sigma(Q_j)$ for the remaining SBs $P_i \setminus \delta$ and $Q_j \setminus \delta$.
   - With the meta-monomorphized traits $\mathcal{T}^{[P_i]}$ and $\mathcal{T}^{[Q_j]}$ fixed, let $\gamma = P_i \overset{\mathcal{T}}{\cap} Q_j \neq \emptyset$ be the set of common SBs with respect to parameters from $\mathcal{T}$. We check if a unifying substitution σ exists such that $\sigma(P_i) \equiv \sigma(Q_j)$ for the remaining SBs $P_i \setminus \gamma$ and $Q_j \setminus \gamma$.

   In essence, both checks fix the common SBs and verify if the remaining, disjoint sets of SBs can be unified, indicating an overlap.
4. The coherence check for specialization bounds must now incorporate actual SBs from the implementing type $\mathcal{S}$ at each call site. Given a well-formed call site:

$$\texttt{spec! \{ s.f(}\underbrace{a_1,\ldots,a_k}_{❶},\underbrace{a_{k+1},\ldots,a_{k+1+l}}_{❷},\underbrace{a_{k+l+2},\ldots,a_m}_{❸},\underbrace{a_{m+1},\ldots a_{m+1+o}}_{❹\ ❺\ ❻}\texttt{);}$$

$$\mathcal{S}^{[P_i]}\texttt{<}\underbrace{\mathbf{B_{m+1}},\ldots,\mathbf{B_n}}_{❹\ ❺\ ❻\ ❼\ ❽\ ❾}\texttt{>; [}\underbrace{\mathbf{B_1},\ldots,\mathbf{B_k}}_{❶},\underbrace{\mathbf{B_{m+1}},\ldots,\mathbf{B_{m+1+o}}}_{❹\ ❺\ ❻}\texttt{];}$$

$$\begin{array}{l}
T_{k+1}\texttt{: }\boldsymbol{\mathcal{T}_{k+1}}\texttt{, ..., }T_{k+1+l}\texttt{: }\boldsymbol{\mathcal{T}_{k+1+l}}\texttt{;}\ ❷\\
T_{m+r+2}\texttt{: }\boldsymbol{\mathcal{T}_{m+r+2}}\texttt{, ..., }T_{m+r+2+s}\texttt{: }\boldsymbol{\mathcal{T}_{m+r+2+s}}\ ❺\ \texttt{\}}
\end{array}$$

   the compiler must identify a unique implementation $(\mathcal{S}^{[P_i]}, \mathcal{T}^{[P_i]}) \in \tilde{\mathrm{I}}$ whose formal SBs P_i match the actual SBs. All type parameters of $\mathcal{S}$ must be provided as actual equality SBs, ensuring the implementing type is fully instantiated at the call site (as Rust lacks higher-rank polymorphism over type constructors). An effective resolution strategy is to first filter candidate implementations based on the equality SBs of $\mathcal{S}^{[P_i]}$, then further refine the selection by matching the trait SBs from $\mathcal{T}^{[P_i]}$.
5. The call site specialization step must now furnish the receiver type with the appropriate type arguments (e.g., `ZST<bool>`). Given the matched implementation $(\mathcal{S}^{[P_i]}, \mathcal{T}^{[P_i]}) \in \tilde{\mathrm{I}}$, the compiler rewrites the call to invoke the method from the meta-monomorphized trait, providing the necessary type arguments for $\mathcal{S}^{[P_i]}$.

$$\begin{array}{l}
\texttt{<}\mathcal{S}^{[P_i]}\texttt{<}\mathbf{B_{m+1}},\ldots,\mathbf{B_{m+1+o}}\texttt{>}\\
\quad\texttt{as }\mathcal{T}^{[P_i]}\texttt{<}T_{k+1},\ldots,T_m,T_{m+r+2},\ldots,T_{m+1+o}\texttt{>>::f(\&s, ...);}
\end{array}$$

3.4 Lifetime Polymorphism with Reference Types

Unsoundness. As noted in §1, the initial `#![feature(specialization)]` gate in Rust was plagued by unsoundness. The core issue was that specialized implementations could inadver-

```
struct ZST;
trait Trait<T, U> { fn f(&self, p: T, u: U); }
const SEVEN: &'static i32 = &7;
#[when(all(T = &str, T: 'a, U = &'a i32))]
impl<'a, T, U> Trait<T, U> for ZST { fn f(&self, p: T, u: U) {} }
#[when(all(T = &str, T: 'a, U = &'b i32))]
impl<'a, 'b, T, U> Trait<T, U> for ZST { fn f(&self, p: T, u: U) {} }
fn main() { let zst = ZST;
    let p: &'static str = "foo";
    spec! { zst.f(p, SEVEN); ZST; [&'static str, &'static i32] }
    spec! { zst.f(p, &7); ZST; [&'static str, &'b i32] } }
```

3 A program with lifetime polymorphism and trait specializations.

tently violate expected lifetime constraints, leading to dangling references and other memory safety vulnerabilities [60]. This problem arises because lifetimes are erased before code generation (specifically, during the MIR-to-LLVM IR lowering), preventing the specialized implementation from being correctly monomorphized with respect to lifetime parameters. In an attempt to mitigate this, Matsakis et al. [61] proposed a more restricted form of specialization, `#![feature(min_specialization)]`, which unfortunately introduced breaking changes for stable Rust. The most robust solution—retaining lifetime information throughout the compilation pipeline—would necessitate a prohibitive "high engineering cost" and significant architectural changes to the compiler [90].

Our approach addresses this challenge by elevating lifetimes to *first-class* specialization parameters.

Example. To illustrate, we adapt the program in Listing 2. The trait `Trait<T>` becomes generic over two types, `T` and `U`; `ZST` is reverted to a monomorphic struct; and a constant `SEVEN` of type `&'static i32` is introduced (cf. Listing 3). The first specialization is governed by the formal SB `all(T = &str, T: 'a, U = &'a i32)`,[12] which constrains `T` and `U` to be references sharing the same lifetime `'a`. The second specialization, `all(T = &str, T: 'a, U = &'b i32)`,[13] constrains them to have distinct lifetimes. In `main`, the first call, `zst.f(p, SEVEN)`, dispatches to the first specialization because both arguments share the `'static` lifetime. The second call, `zst.f(p, &7)`, dispatches to the second specialization, as `p` has a `'static` lifetime while the local reference `&7` has a shorter, anonymous lifetime.

Overview. As the procedure for handling lifetime polymorphism is conceptually equivalent to the predicate polymorphism case (§3.2), we provide a condensed overview. The crucial insight is that lifetimes can be treated as first-class specialization parameters. Lifetime constraints (e.g., `'a: 'b`, `'a = 'static`) are incorporated as atomic predicates within specialization bounds, allowing them to be canonicalized into DNF. The meta-monomorphization procedure generates distinct trait implementations for different lifetime configurations, using the same overlap and coherence verification mechanisms. For instance, the formal SB `all(T = &str, T: 'a, U = &'a i32)` yields a meta-monomorphized trait $\mathcal{T}^{[\texttt{T=\&str,T:'a,U=\&'a i32}]}$ that

[12] The syntaxes `all(T = &str, T: 'a)` and `T = &'a str` are semantically equivalent and interchangeable.

[13] Alternatively, one could use the predicate `all(T = &str, T: 'a, U = &i32, not(U: 'a))` to express that `U` has a lifetime distinct from `'a` without introducing a new lifetime parameter `'b`.

preserves the shared lifetime relationship. Similarly, the SB `all(T = &str, T: 'a, U = &'b i32)` generates $\mathcal{T}^{[\texttt{T=\&str,T:'a,U=\&'b i32}]}$ for cases with distinct lifetimes.

Specialization Bounds Coherence Checking. When resolving method calls involving references, our approach extends SB matching to include lifetime constraints, ensuring coherent dispatch. The resolver must unify type and lifetime parameters simultaneously. For the call `zst.f(p, SEVEN)`, where `p: &'static str` and `SEVEN: &'static i32`, the resolver matches the first specialization's SB, `all(T = &str, T: 'a, U = &'a i32)`, by unifying both lifetimes to `'static`. This produces the substitution `[T` $\mapsto$ `&'static str`, `U` $\mapsto$ `&'static i32`, `'a` $\mapsto$ `'static]`.For the second call, `zst.f(p, &7)`, the local reference `&7` introduces a fresh, shorter lifetime. The resolver then matches the second specialization's SB, `all(T = &str, T: 'a, U = &'b i32)`, yielding the substitution `[T` $\mapsto$ `&'static str`, `U` $\mapsto$ `&'local i32`, `'a` $\mapsto$ `'static`, `'b` $\mapsto$ `'local]`.

Preserving Lifetime Information. Unlike the standard Rust compiler, which erases lifetimes prior to specialization, our meta-monomorphization approach preserves lifetime information throughout the compilation pipeline. This guarantees sound specialization by ensuring each monomorphized instance maintains correct lifetime relationships. The generated implementations retain their lifetime parameters, allowing the borrow checker to verify memory safety at the monomorphized level. For instance, after applying our procedure to Listing 3, the transformed program contains the following specialized traits and implementations:

```rust
struct ZST;
trait Trait<T, U> { fn f(&self, p: T, u: U); }
const SEVEN: &'static i32 = &7;
trait Trait_eq<'a> { fn f(&self, p: &'a str, u: &'a i32); }
trait Trait_noteq<'a, 'b> { fn f(&self, p: &'a str, u: &'b i32); }
impl<'a> Trait_eq<'a> for ZST { fn f(&self, p: &'a str, u: &'a i32) {} }
impl<'a, 'b> Trait_noteq<'a, 'b> for ZST { fn f(&self, p: &'a str, u: &'b i32) {} }
fn main() {
    let zst = ZST; let p: &'static str = "foo";
    <ZST as Trait_eq<'static>>::f(&zst, p, SEVEN);
    <ZST as Trait_noteq<'static, '_>>::f(&zst, p, &7); }
```

This design directly addresses the unsoundness concerns of the original specialization feature by maintaining lifetime precision during code generation, thereby ensuring that specialized implementations cannot violate memory safety invariants.

3.5 Higher-Ranked Polymorphism with Higher-Order Functions

As noted in §2, Rust supports higher-ranked polymorphism via Higher-Ranked Trait Bounds (HRTBs). HRTBs permit the definition of function types that are polymorphic over lifetime parameters, a feature essential for accepting higher-order functions that must operate on references of any lifetime. This capability is particularly valuable for callback patterns and other functional programming constructs where a closure's definition should not unduly constrain the lifetimes of its arguments. However, the interaction between HRTBs and trait specialization remains unexplored in existing Rust implementations, largely due to the aforementioned soundness issues.

Example. Building on the lifetime polymorphism example, Listing 4 demonstrates that our compilation strategy extends naturally to function types universally quantified over lifetimes. The trait `Trait<T, U, V>` now ranges over three type parameters. The key specialization

```
struct ZST;
trait Trait<T, U, V> { fn f(&self, p: T, u: U) -> V; }
const SEVEN: &'static i32 = &7;
#[when(all(T = &str, T: 'b, U = for<'a> fn(T, &'a i32) -> V ))]
impl<'b, T, U, V> Trait<T, U, V> for ZST {
    fn f(&self, p: T, u: U) -> V { u(p, SEVEN) }
}
impl<T, U, V: Default> Trait<T, U, V> for ZST {
    fn f(&self, p: T, u: U) -> V { V::default() }
}
fn main() { let zst = ZST;
    let p: &str = "foo";
    let r: u32 = spec! {
        zst.f(p, |s: &str, n: &i32| { s.len() as u32 + *n as u32 }); ZST;
        [&str, for<'a> fn(&str, &'a i32) -> u32]
    };
    let r2: i32 = spec! { zst.f(p, 2); ZST; [&str, i32] }; }
```

4 A program with higher-ranked polymorphism and trait specializations.

employs the formal SB `all(T = &str, T: 'b, U = for<'a> fn(T, &'a i32) -> V)`, where the `for<'a>` quantifier mandates that the function argument `U` be polymorphic over any lifetime `'a`. This ensures `U` can accept references with any lifetime, not just a specific one. A fallback implementation provides a default behavior for cases that do not match this HRTB specialization.

In the `main` function, the first call passes a closure that conforms to the HRTB specification. This closure can accept an `&i32` reference with any lifetime, including the `'static` lifetime of `SEVEN`. The second call passes an integer instead of a function, causing dispatch to resolve to the fallback implementation.

Overview. Handling HRTBs requires significant modifications to the predicate polymorphism framework (§3.2), particularly with respect to representing and resolving higher-ranked constraints. Our approach treats higher-ranked function types as specialized type constraints. The crucial insight is that HRTB constraints can be encoded as universal quantifications over lifetime parameters within specialization bounds. When a specialization bound contains a `for<'a>` quantifier, our approach generates trait implementations that preserve this higher-ranked nature. The formal SB `all(T = &str, T: 'b, U = for<'a> fn(T, &'a i32) -> V)` results in a meta-monomorphized trait that maintains the universal quantification over `'a` while binding other lifetime relationships.

Specialization Bounds Coherence Checking. Resolving trait method calls with HRTBs requires extending our SB matching algorithm to handle higher-ranked constraints. When the resolver encounters a `for<'a>` quantifier, it must verify that the provided function argument satisfies the constraint for all possible lifetime instantiations. For the call `zst.f(p, |s: &str, n: &i32| ...)`, the resolver must confirm that the closure type `fn(&str, &i32) -> u32` is a subtype of `for<'a> fn(&'b str, &'a i32) -> u32`. This check succeeds because the closure's parameter types do not impose specific lifetime constraints. The resolver produces a substitution `[T` ↦ `&'b str`, `U` ↦ `for<'a> fn(&'b str, &'a i32) -> u32`, `V` ↦ `u32`, `'b` ↦ `'static]`, preserving the higher-ranked nature of `U`.

Preserving Higher-Ranked Information. Unlike traditional compilation approaches that might erase or simplify higher-ranked types during monomorphization, our meta-monomorphization strategy preserves the universal quantification throughout the compilation pipeline. This is essential for maintaining the semantic guarantees of HRTBs, ensuring that specialized implementations can correctly handle function arguments with the required polymorphic behavior. This approach ensures that the higher-ranked polymorphic nature of function arguments is maintained, while enabling precise specialization dispatch based on the structure of the provided closures or function pointers.

3.6 Limitations

Certain classes of programs do not benefit from this approach, particularly those relying on dynamic dispatch or complex type inference.

Existential Polymorphism. As introduced in §1, existential types in Rust are realized via the `impl Trait` or `&dyn Trait` syntax. Our approach does not currently support specialized traits in existential type positions. This limitation stems from a fundamental conflict: existential types conceal concrete type information, whereas our meta-monomorphization strategy depends upon it. When a function returns `impl Trait` or accepts `&dyn Trait` involving a specialized trait, our static, call-site-based approach cannot determine which specialized variant to use because the concrete type is unknown. For `impl Trait` return types, specialization would need to be resolved at the implementation site, but our `spec!` macro demands bounds at the call site. For `&dyn Trait`, the vtable-based dynamic dispatch mechanism is incompatible with our static resolution. Supporting this feature would require a hybrid static-dynamic dispatch mechanism or a method for embedding specialization information within existential types, both of which are interesting directions for future work.

Polymorphic Recursion. Polymorphic recursion, wherein a function calls itself with different type parameters, poses a challenge to our current approach. While Rust supports limited forms of this via trait objects (`Box<dyn Trait>`), our meta-monomorphization strategy struggles with recursive specializations where bounds change across calls. The core issue is that our approach generates a distinct trait implementation for each unique set of specialization bounds. Polymorphic recursion could require an unbounded number of such instantiations within a single execution path. For instance, a recursive function on a nested data structure might require progressively more specific type constraints at each level of recursion, leading to an infinite generation requirement. Addressing this would demand techniques for handling recursive specialization patterns, such as lazy trait generation or cycle detection in the specialization dependency graph. We leave this as an important area for future research.

3.7 Implementation Details

We now provide additional details regarding the implementation of our approach, which has been validated in a Rust software library.

Canonicalization. A critical component of our framework is the canonicalization of predicate formulas into DNF. This transformation ensures that all specialization bounds are represented in a consistent, flat structure, thereby facilitating efficient overlap checking and coherence verification. Given a potentially nested predicate formula $\hat{P}(P_1, \dots, P_x)$, canonicalization proceeds via standard logical transformations. First, De Morgan's laws are applied to push `not` operators to atomic predicates. Next, distributivity rules convert the formula to DNF, where each disjunct represents a complete specialization scenario. Finally, nested `any` predicates are flattened (e.g., `any(any(A, B), C)` becomes `any(A, B, C)`), and redundant clauses are eliminated via subsumption checking. This canonical representation is essential for the efficiency of our overlap detection algorithm.

Coherence. At a given method call site, multiple implementations may be applicable. To resolve such ambiguities, one could adopt the *lattice rule* from the original Rust Specialization RFC [59]. The lattice rule requires that for any two overlapping implementations, a greatest lower bound (GLB)—or *meet*—must exist in the global specialization lattice. This GLB must explicitly handle the intersection, ensuring the compiler can always identify a unique, most-specific implementation. We adopt a more permissive, local-resolution approach. Rather than enforcing global lattice coherence at the definition site, which would require developers to provide exhaustive *intersection* implementations, we resolve dispatch at each specific `spec!` call site. Our system employs a *stratified priority hierarchy* to select the candidate satisfying the most specific conditions, allowing us to support patterns that would be rejected by the strict lattice rule. For example, if a call site matches both T: $\mathcal{T}_1$ and T: $\mathcal{T}_2$ + $\mathcal{T}_3$, the lattice rule would demand a global T: $\mathcal{T}_1$ + $\mathcal{T}_2$ + $\mathcal{T}_3$ implementation. In contrast, our system resolves the call to T: $\mathcal{T}_2$ + $\mathcal{T}_3$, as it is more specific within our partial ordering. By shifting the coherence check from a global property of the trait to a local property of the call site, we provide a more flexible specialization mechanism. If a call remains ambiguous, a compile-time error is issued, prompting the user to refine the local conditions. This resolution is governed by the following partial ordering:

$$
\begin{aligned}
&T = \mathbf{B}_1 \succ T = T_1 \succ T\!: \mathcal{T}_1 + \mathcal{T}_2 \succ T\!: \mathcal{T}_1 \\
&\qquad\qquad \succ T = \mathtt{not}(\mathbf{B}_1) \succ T = \mathtt{not}(T_1) \succ T\!: \mathtt{not}(\mathcal{T}_1 + \mathcal{T}_2) \succ T\!: \mathtt{not}(\mathcal{T}_1)
\end{aligned}
$$

4 Validation

To validate the utility of *meta-monomorphizing specializations*, we conducted an ecosystem-wide analysis of public Rust codebases. Our evaluation quantifies potential improvements in code maintainability and identifies real-world patterns that could benefit from formal specialization mechanisms, as observed in prior work on language tooling and type system reuse [13]. We made a replication package for the experiment publicly available on Zenodo.

Methodology. We developed a static analysis tool by instrumenting the standard Rust compiler with a custom pass, similarly to what is done in [14]. The tool operates on the HIR to reconstruct a custom tree representation for every function, trait, and implementation item. To identify candidate functions for specialization, we employed a two-stage heuristic:

1. **Grouping**: Functions are grouped based on name similarity and signature compatibility (e.g., the same number and types of parameters).
2. **Structural Similarity**: We compute the *tree edit distance* [10, 71, 72] (TED) between the trees within each group using the ZSS algorithm proposed by Zhang and Shasha [98]. Similarity, it is normalized as:

   $$sim(T_1, T_2) = 1 - \frac{\mathrm{TED}(T_1, T_2)}{\max(|T_1|, |T_2|)}$$

   where $|T|$ denotes the number of nodes in tree T. To optimize performance, we bypass pairs where the size ratio $\min(|T_1|, |T_2|)/\max(|T_1|, |T_2|)$ falls below the target threshold, as such pairs cannot mathematically satisfy the similarity criterion.

Dataset. Experiments were executed on an Intel i7-8565U (4C/8T) with 16 GB of RAM, utilizing the `nightly-2025-11-17` toolchain. We applied two similarity thresholds: 90% to capture a broader range of specialization opportunities and 99% to focus on near-identical structures. The dataset comprises representative crates from `crates.io`, spanning various domains and scales (cf. Table 1). To ensure a representative analysis of the Rust ecosystem, we curated a diverse dataset of open-source projects. The selection includes high-traffic crates from `crates.io`, prominent GitHub repositories, and specialized libraries, covering a broad spectrum of architectural patterns. The first three columns of Table 1 summarize the name (along with the link) and total number of functions/traits.

Pattern Identification. Let us define, once and for all, the zero-sized type `struct ZST;` as a type that occupies no memory space. The pattern identification focuses on four prevalent manual specialization patterns currently employed in the ecosystem.

1. The trait provides a version of the function for each type it supports, and the caller is responsible for manually selecting the correct version. For instance:

```
trait Tr<T> { fn fdef(&self, v: T); fn fi32(&self, v: i32); }
impl<T> Tr<T> for ZST { fn fdef(&self, t: T) {} fn fi32(&self, v: i32) {} }
```

2. The trait is manually monomorphized by creating distinct implementations for each type, and the caller manually selects the trait implementation to use. For instance:

```
trait Tr1<T> { fn fdef(&self, v: T); }
trait Tr2 { fn fi32(&self, v: i32); }
impl<T> Tr1<T> for ZST { fn fdef(&self, t: T) {} }
impl Tr2 for ZST { fn fi32(&self, v: i32) {} }
```

3. A distinct function is defined for each type, and the caller manually selects the correct function to call. For instance:

```
fn fdef<T>(x: &MyType, v: T) {} fn fi32(x: &MyType, v: i32) {}
```

4. $\sum$- and $\prod$-types have inherent implementations for each type variant, and the caller manually selects the correct method to call. For instance:

```
impl ZST { fn fdef<T>(&self, v: T) {} fn fi32(&self, v: i32) {} }
```

In all identified patterns, developers must manually dispatch to the appropriate implementation. This approach consistently increases lines of code and maintenance effort. Moreover, each pattern introduces its own form of boilerplate:

- **Redundant Declarations:** Patterns 1, 2, and 4 require developers to write and maintain multiple, nearly identical function or trait declarations, where the only substantive difference is the type signature.
- **Manual Dispatch Logic:** Call sites must implement branching logic, typically via `match` statements on `TypeId`, to select the correct function at runtime. This boilerplate scales linearly with the number of specialized types, compounding complexity.
- **Unsafe Code:** To bridge the gap between the statically unknown generic type and the concrete type required by a specialized function, developers are often forced to employ unsafe operations such as `transmute_copy`.

The following example illustrates the manual dispatch boilerplate common to all these patterns:

```
fn call<T: 'static>(zst: &ZST, v: T) {
    match std::any::TypeId::of::<T>() {
        id if id == std::any::TypeId::of::<i32>() => {
            // SAFETY: We just checked that T is i32
            let v_i32 = unsafe { std::mem::transmute_copy::<T, i32>(&v) };
            // Call the i32 version
        }
        /* Other type arms... */
        _ => { /* Call the default version */ } }
```

It is crucial to emphasize that the value proposition of meta-monomorphization extends far beyond mere LoC reduction. The manual patterns identified are fundamentally constrained by their reliance on nominal type equality checks (`TypeId::of`). This mechanism is inherently deficient, as it lacks support for *predicate polymorphism*. It cannot, for instance, express a condition such as `T=i32` ∨ `T=u32` without duplicating code across multiple `match` arms, further inflating LoC and architectural complexity.

Moreover, these ad hoc solutions are incapable of reasoning about trait bounds (e.g., specializing behavior if a type implements `Clone`) and cannot handle non-static lifetimes, as `TypeId` imposes a `'static` bound. By obviating the need for manual dispatch and unsafe `transmute` operations, a native specialization mechanism yields profound benefits for code safety and maintainability. It replaces fragile, runtime-dependent heuristics with a robust, declarative system, transferring the burden of correctness from the developer to the compiler's type checker and borrow checker. This not only eliminates a significant source of potential memory safety vulnerabilities but also enhances code clarity and simplifies long-term maintenance.

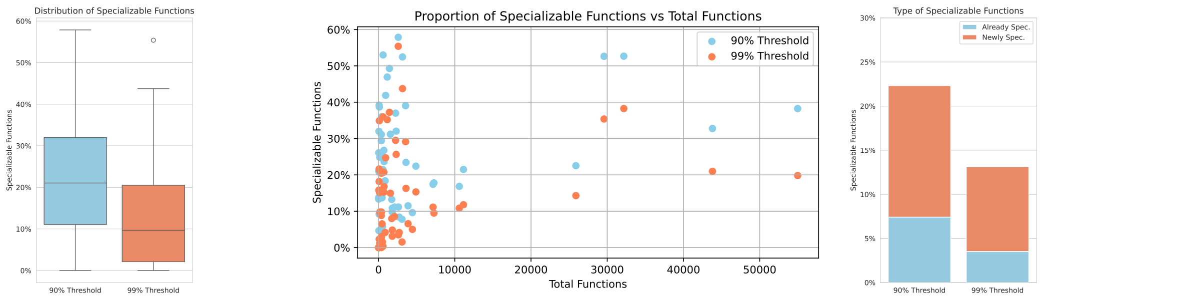
Results. The analysis was conducted using similarity thresholds of 90% and 99%, with the resulting data presented in Table 1. For each project and threshold, we recorded the execution time (in seconds) and peak *resident set size* (RSS) in MB. Additionally, we identified the number of unique specializable functions and traits, reporting both their absolute counts and their respective percentages relative to the project totals. Our analysis reveals that specialization is a pervasive requirement. Across the analyzed projects, we identified numerous instances where specialization could be applied to reduce boilerplate code and improve performance. As shown in Figure 1a, on average more than 20% of functions in the analyzed codebases were found to be specializable at the 90% similarity threshold, with some projects exhibiting even higher proportions.

At the 99% similarity threshold, the average was approximately 10%, indicating that even under stricter similarity requirements, a significant number of functions could benefit from specialization. We observed a positive correlation between project scale and the density of specialization candidates (Figure 1b), suggesting that larger codebases suffer disproportionately from the lack of specialization features.

To evaluate the impact of a specialization implementation compared to Rust's current non-overlapping subset, we categorized specializable functions into two distinct groups:

1. **Already Specializable**: Functions that satisfy our predefined heuristic and also possess a permutation of type parameters that ensures they remain non-overlapping.
2. **Newly Specializable**: Functions that satisfy the heuristic but remain inherently overlapping regardless of parameter permutation, thus requiring a full specialization implementation to be resolved.

Overlaps typically arise from generic type parameters, non-mutually exclusive trait bounds, or identical types across multiple function signatures. For instance, `fn a(x: i32, y: u32)` is



(a) distribution **(b)** correlation with number of functions. **(c)** types

Figure 1 Distribution and types of specializable functions and their correlation with the total number of functions.

	Total		Threshold 90%						Threshold 99%					
					Functions		Traits				Functions		Traits	
	Fns	Trs	*s*	MB	#	%	#	%	*s*	MB	#	%	#	%
syn	11140	129	167	200	2394	21.5%	36	27.9%	200	181	1313	11.8%	30	23.3%
hashbrown	727	34	0.77	106	172	23.7%	10	29.4%	1.12	106	122	16.8%	9	26.5%
bitflags	161	26	0.06	90	40	24.8%	16	61.5%	0.11	90	34	21.1%	15	57.7%
proc-macro2	460	16	3.20	96	63	13.7%	0	0.0%	1.24	94	30	6.5%	0	0.0%
quote	395	32	3.56	89	142	35.9%	4	12.5%	5.27	89	12	3.0%	4	12.5%
base64	84	14	0.04	88	0	0.0%	0	0.0%	0.06	89	0	0.0%	0	0.0%
libc	583	10	208	108	125	21.4%	0	0.0%	293	108	2	0.3%	0	0.0%
getrandom	38	3	0.13	88	8	21.1%	0	0.0%	0.04	88	6	15.8%	0	0.0%
rand	598	50	12.25	121	317	53.0%	15	30.0%	18.10	121	215	36.0%	11	22.0%
indexmap	931	42	2.43	111	390	41.9%	20	47.6%	1.96	110	230	24.7%	17	40.5%
cfg-if	0	0	0.00	51	0	N/A	0	N/A	0.00	51	0	N/A	0	N/A
serde	3141	71	574	329	1648	52.5%	27	38.0%	293	284	1374	43.7%	20	28.2%
itertools	865	33	17.54	140	159	18.4%	3	9.1%	2.53	115	36	4.2%	2	6.1%
autocfg	50	2	0.10	82	16	32.0%	0	0.0%	0.13	83	0	0.0%	0	0.0%
memchr	367	9	6.15	101	108	29.4%	2	22.2%	8.64	101	75	20.4%	2	22.2%
itoa	18	3	0.02	76	0	0.0%	0	0.0%	0.02	76	0	0.0%	0	0.0%
json	1432	49	10.18	119	706	49.3%	13	26.5%	11.31	119	533	37.2%	10	20.4%
thiserror	188	21	0.22	96	18	9.6%	1	4.8%	0.32	96	18	9.6%	1	4.8%
unicode-ident	15	1	0.05	83	2	13.3%	0	0.0%	0.08	83	0	0.0%	0	0.0%
once_cell	106	8	0.17	88	41	38.7%	5	62.5%	0.14	88	37	34.9%	5	62.5%
log	77	7	0.03	76	16	20.8%	3	42.9%	0.06	76	14	18.2%	3	42.9%
heck	23	12	0.03	75	6	26.1%	1	8.3%	0.08	75	0	0.0%	0	0.0%
cc	517	20	0.78	99	30	5.8%	0	0.0%	0.84	98	8	1.5%	0	0.0%
regex	3597	89	150	177	843	23.4%	26	29.2%	52.22	159	586	16.3%	22	24.7%
ryu	43	3	0.41	78	2	4.7%	0	0.0%	0.06	76	0	0.0%	0	0.0%
clap	1730	73	22.69	157	229	13.2%	11	15.1%	13.46	138	138	8.0%	11	15.1%
aho-corasick	699	26	21.76	160	174	24.9%	8	30.8%	5.22	122	145	20.7%	6	23.1%
smallvec	174	35	0.32	95	27	15.5%	5	14.3%	0.17	94	17	9.8%	5	14.3%
strsim	29	3	0.03	83	4	13.8%	0	0.0%	0.04	82	0	0.0%	0	0.0%
parking_lot	363	35	0.88	90	113	31.1%	8	22.9%	0.60	90	32	8.8%	4	11.4%
lazy_static	2	0	0.00	58	0	0.0%	0	N/A	0.00	58	0	0.0%	0	N/A

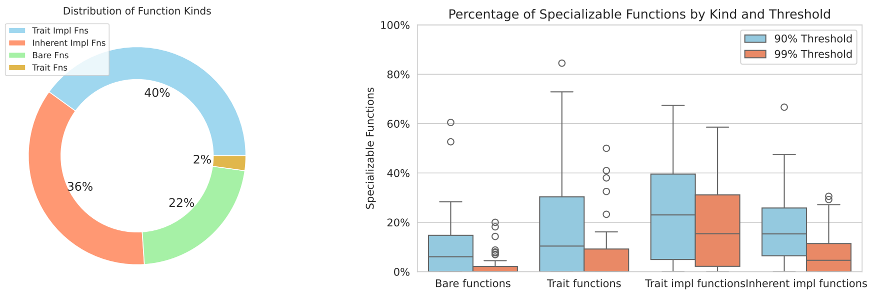
Continued on next page

	Total		Threshold 90%						Threshold 99%					
					Functions		Traits				Functions		Traits	
	Fns	Trs	*s*	MB	#	%	#	%	*s*	MB	#	%	#	%
num-traits	2585	47	53.28	139	1496	57.9%	10	21.3%	48.28	139	1432	55.4%	5	10.6%
socket2	350	12	0.95	94	48	13.7%	1	8.3%	0.36	94	10	2.9%	1	8.3%
semver	87	14	0.38	90	8	9.2%	1	7.1%	0.11	88	2	2.3%	1	7.1%
digest	1142	164	170	108	536	46.9%	43	26.2%	236.6	107	402	35.2%	34	20.7%
either	132	19	0.22	98	28	21.2%	3	15.8%	0.25	98	20	15.2%	2	10.5%
version_check	38	2	0.03	82	8	21.1%	0	0.0%	0.08	82	0	0.0%	0	0.0%
rustix	1809	39	6.64	329	196	10.8%	7	17.9%	6.46	329	87	4.8%	7	17.9%
bytes	695	34	9.64	102	186	26.8%	14	41.2%	11.11	101	106	15.3%	13	38.2%
time	1558	60	123	139	486	31.2%	19	31.7%	172	139	233	15.0%	14	23.3%
url	365	34	0.88	110	8	2.2%	0	0.0%	0.35	108	0	0.0%	0	0.0%
toml	2233	122	26.16	110	826	37.0%	27	22.1%	12.24	109	659	29.5%	20	16.4%
futures	2313	126	14.52	158	741	32.0%	37	29.4%	7.69	158	593	25.6%	30	23.8%
glob	32	7	0.04	84	0	0.0%	0	0.0%	0.01	83	0	0.0%	0	0.0%
tantivy	3871	150	521	253	445	11.5%	24	16.0%	11.14	251	254	6.6%	16	10.7%
tauri	4898	196	469	350	1097	22.4%	34	17.3%	146	350	750	15.3%	26	13.3%
polars	43805	1476	9965	1458	14356	32.8%	388	26.3%	9065	1458	9196	21.0%	293	19.9%
cargo	4441	108	234	413	427	9.6%	18	16.7%	22.23	400	222	5.0%	14	13.0%
bat	356	16	0.24	142	6	1.7%	0	0.0%	0.33	142	0	0.0%	0	0.0%
ripgrep	2096	66	8.37	116	234	11.2%	1	1.5%	3.55	111	179	8.5%	1	1.5%
quiche	2607	110	32.61	172	291	11.2%	16	14.5%	13.10	172	91	3.5%	10	9.1%
influxdb	3547	212	2430	602	1385	39.0%	56	26.4%	1241	601	1034	29.2%	48	22.6%
typst	7260	233	197	376	1293	17.8%	55	23.6%	87.11	375	689	9.5%	42	18.0%
alacritty	2710	72	801	310	227	8.4%	11	15.3%	722	261	112	4.1%	7	9.7%
helix	3080	116	36.18	292	240	7.8%	5	4.3%	10.73	275	47	1.5%	5	4.3%
pueue	389	24	655	1403	58	14.9%	5	20.8%	0.67	176	38	9.8%	4	16.7%
gitoxide	7148	484	617	352	1245	17.4%	36	7.4%	79.75	202	796	11.1%	26	5.4%
texture-synthesis	166	10	0.19	95	2	1.2%	0	0.0%	0.09	95	2	1.2%	0	0.0%
sendme	74	7	2.80	211	29	39.2%	3	42.9%	0.09	191	16	21.6%	1	14.3%
union	29581	1050	13764	854	15570	52.6%	176	16.8%	5467	781	10465	35.4%	90	8.6%
zed	54980	1674	6594	1317	21034	38.3%	256	15.3%	4312	873	10891	19.8%	206	12.3%
ruff	25887	640	1022	1075	5833	22.5%	127	19.8%	471	1074	3698	14.3%	97	15.2%
hyperswitch	32172	826	19738	2635	16947	52.7%	231	28.0%	12533	2642	12309	38.3%	159	19.2%
lapce	1797	62	114	380	176	9.8%	11	17.7%	71.98	367	56	3.1%	5	8.1%
nushell	10592	330	1155	2716	1783	16.8%	61	18.5%	132	325	1148	10.8%	51	15.5%

Table 1 Experimental results of the ecosystem-wide analysis. For each analyzed crate, the table reports the total number of functions (**Fn**) and traits (**Tr**), followed by performance metrics and specialization candidates identified at two similarity thresholds (90% and 99%). Metrics include execution time in seconds (***s***), peak memory usage in megabytes (**MB**), and the absolute number (#) and percentage (%) of specializable functions and traits.

considered "already specializable" in relation to `fn b<T>(x: T, y: u32)`, as the latter can be permuted into a non-overlapping form. A critical finding is the delta between already specializable and newly specializable functions. Full stable specialization triples the available candidates compared to current non-overlapping rules. At the 90% threshold, 67% of identified candidates require stable specialization to be implemented natively (Figure 1c). This confirms that current language limitations force developers into the suboptimal patterns identified above.

Subsequently, we sought to determine which of the patterns identified earlier were most prevalent in the analyzed codebases. To this end, we first classified the functions into four



(a) Functions by Kind **(b)** Specializable Functions by Kind.

Figure 2 Distribution of functions and specializable functions by kind.

distinct categories based on their structure: bare functions (not associated with any trait or impl), trait functions (defined within a trait), trait impl functions (defined within a trait impl block), and inherent impl functions (defined within an inherent impl block). As shown in Figure 2a, we found that the majority of functions in the analyzed codebases were trait impl functions, followed by inherent impl functions and bare functions, with trait functions being the least common by a substantial margin. As Figure 2b demonstrates, the group with the majority of specializable functions consisted of trait impl functions, indicating that these functions were not only the most common but also the most likely to benefit from stable specialization. In particular, the most common patterns associated with trait impls are the first two patterns identified earlier (manual monomorphization via distinct trait implementations and via multiple trait methods), suggesting that many developers resort to these approaches to achieve specialization in their code. As noted earlier, these two patterns would benefit most from specialization features, as both would see a substantial reduction in boilerplate code and complexity.

Discussion. While the data suggests significant benefits, several factors merit consideration. Specialization is not a universal solution; the trade-off between performance gains and binary size or compile-time complexity must be evaluated on a case-by-case basis. Our tree-based similarity metric relies on naming and structure. Although effective, it may yield false positives in cases of coincidental structural similarity or false negatives where the logic is semantically identical but structurally divergent. The prevalence of these patterns does not necessarily indicate "poor code" but rather reflects the lack of expressive power in the current trait system when dealing with overlapping implementations.

Crucially, our meta-monomorphization approach constitutes a practical solution that is immediately available, in contrast to Rust's native specialization, which has remained unstable on the nightly channel for years due to unresolved soundness concerns [90]. Our data reveals that 67% of specialization candidates require full overlapping support, a capability absent from Rust's current non-overlapping specialization subset. Meta-monomorphization fills this gap without compiler changes, eliminating reliance on `unsafe` operations such as `transmute_copy`, enabling predicate polymorphism over trait bounds beyond `TypeId`-based dispatch, and supporting non-`'static` lifetimes while remaining fully compatible with standard compiler optimizations. By shifting specialization to the metaprogramming layer, developers gain immediate access to expressive specialization patterns that would otherwise remain indefinitely blocked.

In conclusion, the data demonstrates that specialization would significantly reduce boilerplate and formalize common architectural workarounds, thereby enhancing the overall robustness of the Rust ecosystem.

5 Threats to Validity

We organize our discussion following the taxonomy of Wohlin et al. [97]'s taxonomy.

Construct Validity. Our study relies on specific metrics to evaluate the effectiveness of our approach. If these metrics do not accurately capture the constructs we intend to measure, the validity of our conclusions could be threatened. To mitigate this risk, we carefully selected metrics that are widely accepted in the research community and relevant to our study's objectives. The criteria used to determine the similarity between code snippets may not fully capture the nuances of code functionality and intent. Hence, our similarity assessments might not reflect true equivalence in behavior. We based our similarity criteria on established practices in code analysis and validated them through preliminary experiments to mitigate this threat.

Internal Validity. Our approach relies on certain assumptions about the structure of HIRs generated by the Rust compiler. If these assumptions do not hold for all codebases, the validity of our results could be affected. The mitigation strategy involved thorough testing of our method across a variety of Rust projects to ensure that our assumptions were valid in practice.

External Validity. Our evaluation is based on 65 open-source projects from GitHub and `crates.io`. While these projects cover a broad spectrum of real-world software, they may not capture the full variability of proprietary or industrial codebases. However, many of the analyzed projects are widely used in production and serve as dependencies for industrial systems. This increases our confidence that the results generalize beyond purely academic or hobbyist software. The selection of projects may introduce bias if certain types of software or development practices are over represented. To mitigate this issue, we systematically included a diverse set of projects by selecting repositories of varying sizes, domains, and activity levels.

Conclusion Validity. Our quantitative results depend on the accuracy of our data collection and analysis methods. Errors in data extraction, measurement, or statistical analysis could lead to incorrect conclusions. Nonetheless, we employed automated tools for data collection and analysis to minimize human error. Additionally, we performed multiple runs of our experiments to ensure the consistency of the results.

6 Related Work

Parametric Polymorphism & Monomorphization. Our work is grounded in the tradition of parametric polymorphism [18, 76, 93, 36], as formalized in System F [32, 77, 16] and Hindley-Milner type systems [40, 64]. To eliminate abstraction overhead over algebraic data types [55, 89, 9], numerous languages—including C++ [84, 91, 83, 2], Rust [62], Go [35], MLton [20, 94], and Futhark [41]—employ monomorphization. Recent formal treatments [57] have extended this concept to encompass existential and higher-rank polymorphism. While conventional optimizing compilers [1, 48, 27, 66] utilize interprocedural analyses and procedure cloning [25, 26, 30, 42] to enable optimizations like inlining [80, 19, 4] and SSA-based transformations [17, 95, 96, 29, 51], our approach diverges by shifting the specialization process to the metaprogramming stage. This allows us to reuse existing optimization passes without necessitating intrusive compiler modifications.

Ad Hoc Polymorphism & Specialization. Beyond zero-cost parametricity, specialization is a vital mechanism for ad hoc polymorphism (e.g., traits, interfaces), enabling the exploitation of hardware idioms (SIMD) or optimized algorithms [3]. C++ facilitates this through explicit and partial template specialization [84, 91, 6], while Haskell utilizes the `SPECIALIZE` pragma [74] over its dictionary-passing implementation of type classes [75, 36, 85]. In Rust, the stabilization of a native specialization feature remains deferred due to unresolved soundness and coherence concerns [59, 60, 61, 79, 90]. In contrast, our "meta-monomorphization" technique preserves developer control through code generation (macros), thereby circumventing the need to extend the language's trait solver.

Coherence, Safety & Limits. The specialization of interfaces introduces significant challenges concerning coherence [76, 44, 28, 85], overlapping instances [86, 85], and orphan rules.[4] By resolving specialization choices via explicit predicates during macro expansion, we sidestep the pitfalls of implicit global resolution. Nevertheless, our work acknowledges and respects the theoretical limitations inherent in specializing polymorphic recursion and existential types [31, 47, 43, 38, 49, 65, 54, 70, 11, 37, 78]. Consequently, we target first-order programs and specific higher-ranked patterns where such issues do not arise.

Metaprogramming & Ecosystems. Existing mechanisms such as Scala's `@specialized` [69] annotation and Java's Project Valhalla[6] have attempted to mitigate the effects of type erasure and boxing, but they often introduce challenges related to code bloat or runtime complexity. Leveraging modern metaprogramming paradigms [56, 81, 15, 23, 39, 52, 24, 21], our framework emits specialized implementations before the type-checking phase. This design ensures compatibility with mature compiler pipelines while offering a practical path to specialization in languages that lack stable, built-in support. Our primary contribution, therefore, is the strategic shift of specialization to compile-time metaprogramming. This approach yields deterministic, type-checked specialized code without modifying the host compiler or its trait solver. While acknowledging the known limits of polymorphic recursion and existential quantification, we focus on first-order programs and a restricted set of rank-1 and rank-2 patterns where explicit, predicate-driven selection results in predictable code size and performance characteristics.

7 Conclusion

In this work, we have introduced *meta-monomorphizing specializations*, a novel framework for achieving zero-cost specialization by leveraging compile-time metaprogramming. By encoding specialization constraints as type-level predicates, our approach enables deterministic and coherent dispatch without requiring modifications to the host compiler or contending with the complexities of overlapping instances. We have provided a formal treatment of our method, covering first-order, predicate-based, and higher-ranked trait bound (HRTB) specializations, complete with robust support for lifetime polymorphism. Our analysis of public Rust codebases reveals that specialization is a prevalent and vital optimization strategy. Meta-monomorphization offers a principled alternative to common, often unsafe, workarounds, ultimately yielding more idiomatic, maintainable, and performant code.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.
2. Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, February 2001.
3. Johnathan Alsop, Weon Taek Na, Matthew D. Sinclair, Samuel Grayson, and Sarita Adve. A Case for Fine-grain Coherence Specialization in Heterogeneous Systems. *ACM Transactions on Architecture and Code Optimization*, 19(3):41:1–41:26, September 2022.
4. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive Inlining. In A. Michael Berman, editor, *Proceedings of the 18th Conference on Programming Language Design and Implementation (PLDI'97)*, pages 134–145, Las Vegas, NV, USA, June 1997. ACM.
5. Franz Baader, Ralf Molitor, and Stephan Tobies. Tractable and Decidable Fragments of Conceptual Graphs. In William M. Tepfenhart and Walling R. Cyre, editors, *Proceedings of the 7th International Conference on Conceptual Structures (ICCS'99)*, LNCS 1640, pages 480–493, Blackburg, VA, USA, July 1999. Springer.
6. Bruno Bachelet, Antoine Mahul, and Loïc Yon. Template Metaprogramming Techniques for Concept-Based Specialization. *Scientific Programming*, 21(1-2):43–61, January 2013.
7. Patrik Backhouse, Roland Carland Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic Programming: An Introduction. In S. Doaitse Swierstra, Pedro Rangel Henriques, and José Nuno Oliveira, editors, *Proceedings of the 3rd International School on Advanced Functional Programming (AFP'98)*, LNCS 1608, pages 28–115, Braga, Portugal, September 1998. Springer.
8. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
9. Jan A. Bergstra and John V. Tucker. Equational Specifications, Complete Term Rewriting Systems, and Computable and Semicomputable Algebras. *Journal of ACM*, 42(6):1194–1230, November 1995.
10. Philip Bille. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science*, 337(1-3):217–239, June 2005.
11. Richard Bird and Lambert Meertens. Nested Datatypes. In *Proceedings of the 4th International Conference on Mathematics of Program Construction (MPC'98)*, LNCS 1422, pages 52–67, Marstrand, Sweden, June 1998. Springer.
12. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races And Deadlocks. In Satoshi Matsuoka, editor, *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, Seattle, WA, USA, November 2002. ACM Press.
13. Federico Bruzzone, Walter Cazzola, and Luca Favalli. Code Less to Code More: Streamlining Language Server Protocol and Type System Development for Language Families. *Journal of Systems and Software*, 231, January 2026. doi:10.1016/j.jss.2025.112554.
14. Federico Bruzzone, Walter Cazzola, and Luca Favini. Prioritizing configuration relevance via compiler-based refined feature ranking, 2026. URL: https://arxiv.org/abs/2601.16008, arXiv:2601.16008.
15. Eugene Burmako. Scala Macros: Let Our Powers Combine! On How Rich Syntax and Static Types Work with Meta-Programming. In *Proceedings of the 4th Workshop on Scala (SCALA'13)*, Montpellier, France, July 2013. ACM.
16. Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System F-Omega with Equirecursive Types for Datatype-Generic Programming. In Rupak Majumdar, editor, *Proceedings of the 43rd Symposium on Principles of Programming Languages (POPL'16)*, pages 30–43, St. Petersburg, FL, USA, January 2016. ACM.
17. David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation. In Richard L. Wexelblat, editor, *Proceedings of the Sigplan Symposium on Compiler Construction (SCC'86)*, pages 152–161, Palo Alto, CA, USA, June 1986. ACM.

18 Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

19 John Cavazos and Michael F. P. O'Boyle. Automatic Tuning of Inlining Heuristics. In Jeff Kuehn and Wes Kaplow, editors, *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*, pages 14–14, Seattle, WA, USA, November 2005. IEEE.

20 Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-Directed Closure Conversion for Typed Languages. In Gert Smolka, editor, *Proceedings of the 9th European Symposium on Programming (ESOP'00)*, LNCS 1782, pages 56–71, Berlin, Germany, March 2000. Springer.

21 Adam Chlipala. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In Alex Aiken, editor, *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI'10)*, pages 122–133, Toronto, Canada, June 2010. ACM.

22 David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In Craig Chambers, editor, *Proceedings of 13th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 48–64, Vancouver, BC, Canada, October 1998. ACM.

23 William D. Clinger and Jonathan Rees. Macros That Work. In David Wise, editor, *Proceedings of the 18th Symposium on Principles of Programming Languages (POPL'91)*, pages 155–162, Orlando, FL, USA, January 1991. ACM.

24 William D. Clinger and Mitchell Wand. Hygenic Macro Technology. In Guy L. Steele Jr and Richard P. Gabriel, editors, *Proceedings of the 4th History of Programming Languages Conference (HOPL'21)*, pages 1–110, Virtual, June 2021. ACM.

25 Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In Carl K. Chang, editor, *Proceedings of the 4th International Conference on Computer Languages (ICCL'92)*, pages 95–105, Oakland, CA, USA, April 1992. IEEE.

26 Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A Methodology for Procedure Cloning. *Computer Languages*, 19(2):105–117, April 1993.

27 Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, November 2022.

28 Pierre-Louis Curien and Giorgio Ghelli. Coherence of Subsumption, Minimum Typing and Type-Checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, March 1992.

29 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

30 Dibyendu Das. Function Inlining Versus Function Cloning. *Sigplan Notices*, 38(6):23–29, June 2003.

31 Richard A. Eisenberg. Levity Polymorphism. In Martin Vechev, editor, *Proceedings of the 38th Conference on Programming Language Design and Implementation (PLDI'17)*, pages 525–539, Barcelona, Spain, June 2017. ACM.

32 Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Phd thesis, Université Paris VII, Paris, France, June 1972.

33 Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

34 Jean-Yves Girard, Yves Lafont, and Laurent Regnier. *Advances in Linear Logic*. Cambridge University Press, July 1995.

35 Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight Go. In David Grove, editor, *Proceedings of the 35th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20)*, pages 1–29, Chicago, IL, USA, November 2020. ACM.

36 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

37 J.J. Hallett and Kfoury. Assef J. Programming Examples Needing Polymorphic Recursion. *Electronic Notes in Theoretical Computer Science*, 136:57–102, July 2005.

38 Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

39 David Herman and Mitchell Wand. A Theory of Hygieninc Macros. In Sophia Drossopoulou, editor, *Prooceedings of the 17th European Conference on Programming Languages and Systems (ESOP'08)*, LNCS 4960, pages 48–62, Budapest, Hungary, March/April 2008. Springer.

40 Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the America Mathematical Society*, 146:29–60, December 1969.

41 Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. High-Performance Defunctionalisation in Futhark. In MichałPałka and Magnus Myreen, editors, *Proceedings of the International Symposium on Trends in Functional Programming (TFP'18)*, LNCS 11457, pages 136–156, Gothenburg, Sweden, June 2018. Springer.

42 Robert Husák, Jan Kofroň, Jakub Míšek, and Filip Zavoral. Using Procedure Cloning for Performance Optimization of Compiled Dynamic Languages. In Hans-Georg Fill and Marten van Sinderen, editors, *Proceedings of the 17th International Conference on Software Technologies (ICSOFT'22)*, pages 175–186, Lisbon, Portugal, 2022. ScitePress.

43 Shengyi Jiang, Chen Cui, and Bruno C. d. S. Oliveira. Bidirectional Higher-Rank Polymorphism with Intersection and Union Types. In Armando Solar-Lezama, editor, *Proceedings of the Symposium on Principles of Programming Languages (POPL'25)*, pages 2118–2148, Denver, CO, USA, January 2025. ACM.

44 Mark P. Jones. Coherence for Qualified Types. Research Report YALEU/DCS/RR-989, Yale University, New Haven, CT, USA, September 1993.

45 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. In Andrew D. Gordon, editor, *Proceedings of the 44th Symposium on Principles of Programming Languages (POPL'17)*, pages 66:1–66:34, Paris, France, January 2017. ACM.

46 Andrew Kennedy and Claudio V. Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In Richard P. Gabriel, editor, *Proceedings of 19th ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05)*, pages 21–40, San Diego, CA, USA, October 2005. ACM.

47 Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI01)*, pages 1–12, Snowbird, Utah, USA, June 2001.

48 Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers Inc., October 2001.

49 Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

50 Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language.* No Starch Press, third edition, February 2026.

51 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial Dead Code Elimination. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Proceedings of the 15th Annual Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–158, Orlando, FL, USA, June 1994. ACM.

52 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic Macro Expansion. In William L. Scherlis, John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the 3rd Conference on LISP and Functional Programming (LFP'86)*, pages 151–161, Cambridge, MA, USA, August 1986. ACM.

53 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In Michael D. Smith, editor, *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, San José, CA, USA, March 2004. IEEE.

54 Konstantin Läufer. Type Classes with Existential Types. *Journal of Functional Programming*, 6(3):485–518, May 1996.

55 Daniel J. Lehmann and Michael B. Smyth. Algebraic Specification of Data Types: A Synthetic Approach. *Journal of Mathematical Systems Theory*, 14(2):97–139, December 1981.

56 Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. *ACM Computing Surveys*, 52(6), October 2019.

57 Matthew Lutze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. The Simple Essence of Monomorphization. In Shriram Krishnamurthi and Sukyoung Ryu, editors, *Proceedings of the 40th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'25)*, pages 1015–1041, Singapore, October 2025. ACM.

58 José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing Generics Is Easy! In John P. Gallagher and Janis Voitländer, editors, *Proceedings of the 19th Workshop on Partial Evaluation and Program Manipulation (PEPM'10)*, pages 33–42, Madrid, Spain, January 2010. ACM.

59 Nicholas D. Matsakis. Specialization. RFC 1210, June 2015. https://rust-lang.github.io/rfcs/1210-impl-specialization.html.

60 Nicholas D. Matsakis. Specialization. Discussion on RFC 1210, June 2015. https://github.com/rust-lang/rfcs/pull/1210.

61 Nicholas D. Matsakis. Maximally Minimal Specialization: Always Applicable `impls`. Blog Post, February 2018. https://smallcultfollowing.com/babysteps/blog/2018/02/09/maximally-minimal-specialization-always-applicable-impls/.

62 Nicholas D. Matsakis and Felix S. Klock. The Rust Language. *ACM SIGAda Letters*, 34(3):103–104, October 2014.

63 John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine (Part I). *Communications of the ACM*, 3(4):184–195, April 1960.

64 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

65 John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

66 Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, first edition, August 1997.

67 David R. Musser and Alexander A. Stepanov. Generic Programming. In Patrizia M. Gianni, editor, *Proceedings of the 13th International Symposium on Symbolic and Algebraic Computation (ISAAC'88)*, LNCS 358, pages 13–25, Rome, Italy, July 1988. Springer.

68 Martin Odersky. Observers for Linear Types. In Bernd Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP'92)*, LNCS 582, pages 390–407, Rennes, France, February 1992. Springer.

69 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Aritma Press, 2008.

70 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, first edition, June 1999.

71 Mateusz Pawlik and Nikolaus Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. In José Blakely, Joseph M. Hellerstein, Nick Koudas, Wolfgang Lehner, Sunita Sarawagi, and Uwe Röhm, editors, *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB'12)*, volume 5, pages 334–345, Instanbul, Turkey, January 2011. ACM.

72 Mateusz Pawlik and Nikolaus Augsten. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems*, 40(1):3:1–3:40, March 2015.

73 Francis Jeffry Pelletier and Allen Hazen. Natural Deduction Systems in Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Stanford University, October 2021.

74 Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

75 Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type Classes: An Exploration of the Design Space. In John Launchbury, editor, *Proceedings of the 2nd Workshop on Haskell (Haskell'97)*, pages 1–16, Amsterdam, The Netherlands, June 1997. ACM.

76 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002.

77 John C. Reynolds. Towards a Theory of Type Structure. In Bernard J. Robinet, editor, *Proceedings of 1974 Programming Symposium*, LNCS 19, pages 408–423, Paris, France, April 1974. Springer.

78 Eric S. Roberts. *Thinking Recursively*. John Wiley and Sons, Inc, first edition, April 1986.

79 Rust Project Developers. Rust release channels. https://doc.rust-lang.org/book/appendix-07-nightly-rust.html, 2024.

80 Robert W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. *Communications of the ACM*, 20(9):647–654, September 1977.

81 Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. In Manuel Chakravarty, editor, *Proceedings of the 6th Workshop on Haskell (Haskell'02)*, pages 1–16, Pittsburg, PA, USA, October 2002. ACM.

82 Christopher Strachey. Fundamental Concepts in Programming Languages. *Journal Higher-Order and Symbolic Computation*, 13(1-2):11–49, April 2000.

83 Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, March 1994.

84 Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, fourth edition, July 2005.

85 Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, November 2005.

86 Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with Type Equality Coercions. In George Necula, editor, *Proceedings of the International Workshop on Types in Languages Design and Implementation (TLDI'07)*, pages 53–66, Nice, France, January 2007. ACM.

87 The Rust Project Developers. The rustonomicon, n.d. https://doc.rust-lang.org/nomicon/.

88 Laurence Tratt. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems*, 30(6):31:1–31:40, October 2008.

89 David A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Functional Programming Languages and Computer Architecture (FPCA'85)*, LNCS 201, pages 1–16, Nancy, France, September 1985. Springer.

90 Aaron Turon. Shipping Specialization: A Story of Soundness. Blog Post, July 2017. https://aturon.github.io/blog/2017/07/08/lifetime-dispatch/.

91 David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, November 2002.

92 Philip Wadler. Linear Types Can Change the World! In Manfred Broy and Cliff B. Jones, editors, *Proceedings of the 2nd Working Conference on Programming Concepts and Methods (IFIP'90)*, pages 561–582, Sea of Galilee, Israel, April 1990. North-Holland.

93 Philip Wadler and Stephen Blott. How to Make Ad-Hoc Polymorphism Less Ad-Hoc. In *Proceedings of the 16th Symposium on Principles of Programming Languages (POPL'88)*, pages 60–76, Austin, TX, USA, January 1988. ACM.

94 Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and François Pottier, editors, *Proceedings of the Workshop on ML (ML'06)*, page 1, Portland, OR, USA, 2006. ACM.

95 Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

96 Mark N. Wegman and Frank Kenneth Zadeck. Constant Propagation with Conditional Branches. In Mary S. Van DEusen, Zvi Galil, and Brian K. Reid, editors, *Proceedings of the*

12th Symposium on Principles of Programming Languages (POPL'85), pages 291–299, New Orleans, LA, USA, January 1985. ACM.

97 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

98 Kaizhong Zhang and Dennis Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *Journal on Computing*, 18(6):1245–1262, December 1989.