Stageleft: Multi-stage Programming in Standard Rust

Shadaj Laddad*
Amazon Web Services
shadajl@amazon.com

Mingwei Samuel†
Amazon Web Services
mingwes@amazon.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

Abstract

Rust has emerged as a popular systems language with growing interest in metaprogramming, yet it lacks staging support—developers must write unsafe, untyped macros instead. We present Stageleft, a library that brings type-safe staged programming to standard Rust without compiler modifications. Stageleft ensures hygienic code generation through ahead-of-time AST analysis, and handles free variables via a trait system that respects Rust's ownership rules. Stageleft demonstrates that staging can be practical and safe in Rust, enabling domain-specific optimizations while maintaining familiar developer interfaces.

CCS Concepts: • **Software and its engineering → Source code generation**; **Language features**.

Keywords: Rust, multi-stage programming, macros

ACM Reference Format:
Shadaj Laddad, Mingwei Samuel, and Joseph M. Hellerstein. 2026. Stageleft: Multi-stage Programming in Standard Rust. In *Proceedings of the 25th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '26), June 29, 2026, Brussels, Belgium.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3814885.3816414

1 Introduction

A major focus of modern languages is *zero-cost abstraction*, which aims to eliminate the performance impact of code reuse. Languages like Rust use specialization and compile-time memory management to enable such abstractions. But these techniques only go so far, as the structure of the source code still has an impact on the compiled binary—it determines function boundaries, execution order, and optimization opportunities. This leads to performance penalties when using abstractions that are not well aligned with low-level execution, such as dataflow programming.

Multi-stage programming [15, 16] is a powerful metaprogramming technique that aims to address this. Instead of directly compiling the source code to a runtime binary, staged programming compiles code using several (typically two) *stages*. When a program that uses staging is compiled and run, it does not directly execute the defined computation; instead, it generates a new source file containing the computation. This generated code is then compiled (the second stage) and executed to yield the final result.

Staged programming decouples the abstractions and interfaces exposed to developers from the structure of the compiled runtime code, all with minimal impact to the end-user APIs. Multi-stage programming has been used extensively in deep learning engines [1, 2, 9], query optimizers [11], and stream processing [4, 8] to extract bare-metal performance without impacting the high-level abstractions.

Some languages support staged programming out of the box with built-in constructs. For example, MetaOCaml [14] is a fork of the OCaml language that offers built-in staging mechanisms. Similarly, the Scala language has a rich metaprogramming system that enables staged programming [10, 12, 13]. These languages have deep integration with the compiler to make staging a first-class construct. Yet most languages, even those with metaprogramming capabilities, still do not support staged programming.

Rust offers a powerful macro system that allows developers to write code generators, but does not support staged programming. The Rust macro system requires users to manipulate untyped blobs of tokens. While this is fine for experts writing specialized code generators, it is impractical for developers to write application logic with such mechanisms. Staged programming is only practical when the high-level APIs have well-typed signatures and appear as regular functions from the perspective of the user.

In this paper, we present *Stageleft*, a staged programming library for Rust[1]. Stageleft enables library authors to generate high-performance code while preserving a familiar, type-safe API for users. It uses Rust's existing macro system under the hood, which means that it can be used in any Rust program without requiring special support from the compiler. Stageleft has been used extensively to power zero-cost distributed programming in the Hydro framework [6].

Stageleft provides strong static guarantees that the generated code is well-typed, and uses Rust's type system in novel ways to enable use across domains like distributed systems. Programs written in Stageleft can be consumed from other

*Work done while at UC Berkeley.
†Work done while at Sutter Hill Ventures.

GPCE '26, Brussels, Belgium

ACM ISBN 979-8-4007-2718-4/2026/06
https://doi.org/10.1145/3814885.3816414

[1] https://github.com/hydro-project/stageleft

Rust code as macros, or can generate standalone Rust crates that can be compiled and run independently.

The primary constructs for staged programming are *quoting* and *splicing*, which have similar behavior to their typical metaprogramming forms but typically come with stronger typing guarantees. Stageleft provides these with a Rust macro that can be used to quote Rust expressions into a value that captures the type of the quoted code, which can then be spliced into another quoted expression to compose code blocks. Libraries are free to manipulate the quoted code using standard Rust metaprogramming libraries like syn [19] to generate runtime logic.

A unique aspect of Stageleft is that it extensively leverages Rust traits to define interfaces for splicing values. While Stageleft provides built-in implementations for composing quotes and splicing constants, it also allows library authors to define implementations for their own types. This enables libraries to create special values that behave differently in a quoted context, such as for accessing metadata that is only available in the final generated code.

A key goal of staged programming is that if the staged code compiles, the generated code will also compile. Because Stageleft is a library rather than a compiler extension, this guarantee holds relative to a set of trusted components, and with restrictions on the quoted code. *This is a compromise.* Stageleft may admit code that fails to compile when spliced, or reject quoted code that would otherwise compile.

We accept this tradeoff in exchange for immediate utility for the broader Rust community, since Stageleft can be used with the standard stable Rust toolchain. We have found that Stageleft's limitations are manageable in practice with well-documented workarounds. We discuss the source of these limitations and identify opportunities for tighter integration with the Rust compiler to provide stronger guarantees.

In summary, we make the following contributions:

- We present a construct for quoting and splicing code in standard Rust that preserves type safety and enables seamless use by libraries (Section 3)
- We develop a mechanism for hygienic expansion without special support from the Rust compiler (Section 4)
- We discuss how Stageleft allows quoted code to reference free variables and how library authors can customize their expansion (Section 5)
- We show how staged programs can be exposed as macros or standalone crates (Section 6)
- We evaluate Stageleft through a case study of the Hydro distributed programming framework and a microbenchmark demonstrating the performance impact of staged stream fusion (Section 7)

2 Metaprogramming in Rust

Before we dive into Stageleft, we first need to understand the landscape of existing metaprogramming techniques in Rust. Rust offers a pair of built-in metaprogramming features: *declarative macros* and *procedural macros*. Declarative macros are a simple way to define code generation rules using a pattern-matching syntax. They are typically used for simple code generation tasks, such as generating boilerplate code or implementing common traits.

Procedural macros are closer to staged programming. They allow developers to write custom code-generation logic using Rust itself, without any limitations on the macro logic or control flow. Procedural macros are defined as Rust functions `fn(TokenStream) -> TokenStream`: they receive the tokens written at the call site and return new tokens that replace the macro invocation. The compiler substitutes the returned tokens into the program and then proceeds with typechecking. Libraries can expose procedural macros using several syntactic constructs: as a function-like macro or as an attribute. In this paper, we will focus on function-like macros, which consume and emit Rust expressions.

To build a procedural macro, the developer must create a separate Rust crate with the macro logic, which is dynamically loaded by the compiler. Because the Rust compiler provides a very minimal set of APIs for interacting with tokens, developers often use third-party libraries like `syn` [19] to parse the token stream into a Rust AST and manipulate it, and the `quote` crate to construct new token streams from templates. When a macro is used in a program, the compiler invokes the macro implementation *before typechecking* the program. This offers significant flexibility—the macro can introduce new types and functions—but it also means that the macro does not have access to type or symbol information.

Rust tokens are untyped, which is a significant difference from staged programming. With staging, quoted code may be stored as tokens under the hood, but the container captures the expression's type. When composing quoted snippets, this type information is used to ensure that the generated code is well-typed. In contrast, Rust macros emit arbitrary strings of Rust syntax, so macro authors are responsible for ensuring that the generated code will compile.

The other major distinction from staging is the symbols that can be referenced from quoted code. Rust procedural macros are *not hygienic*, and can reference any symbol in the scope of the macro invocation. A further effect is that quoted code in the macro implementation cannot reference symbols defined alongside the macro such as a helper function. Instead, the macro must generate all code under the assumption that it will be pasted into a different crate.

This is problematic for a library that wants to expose a staged interface, since developers often reference local symbols in closures passed to a library API. Stageleft addresses this by ensuring that quoted code is hygienic, and by providing ways for quoted code to safely reference local symbols. This is done by applying a lightweight program analysis to identify external symbols, and a trait system that generates the appropriate code to resolve these symbols in spliced code.

3 Quoting and Splicing

Developers using Stageleft will spend most of their time interacting with the `q!` macro, which is the primary interface for quoting a Rust expression in a type-safe manner. The `q!` macro is a function-like macro that takes a Rust expression as input and produces a quoted expression. The quoted expression can then be spliced into another quoted expression using the `q!` macro, or into a raw set of tokens using the `splice` method. In this section, we will discuss the basics of quoting and splicing in Stageleft, including how we built type-safe quoting on top of Rust's existing macro system.

3.1 Quoting Expressions

When quoting a piece of code in Stageleft, the quoted code is returned as a value that implements the `Quoted` trait, which captures the tokens in a type-safe container that tracks the type of the quoted expression. The only APIs available on this trait are for splicing the expression. Instead of using a struct type for quoted values, Stageleft uses traits to enable library authors to define their own data types with custom splicing behavior. We show the core definition of the `Quoted` trait in Figure 1. Our definitions vary slightly from the open-source implementation for simplicity; we omit lifetimes that enable more use cases but do not affect overall correctness.

```
pub trait Quoted<T> {
    fn splice(self) -> TokenStream;
}
```

Figure 1. The `Quoted` trait captures quoted expressions.

The primary way to create a quoted value is with the `q!` macro, a function-like procedural macro which takes a Rust expression as input and produces a quoted expression. Unlike the existing `quote` macro provided by Rust, the `q!` macro returns a value that captures the expression type. To achieve this, we need to expand the macro such that the provided expression is typechecked, but *not run*. For Rust's typechecker to infer the expression's type, the expression must show up as a return value. Our trick is to generate a closure whose return type will capture the quoted expression's type, while storing the tokens of the expression into a string as a side effect when it is invoked.

Consider a simple example of quoting an arithmetic Rust expression. The quoting macro stores the AST of the expression in a string, and then emits the expression itself for typechecking purposes in an unreachable branch. Because both branches of the conditional must return a value of the same type, we return an uninitialized value using `MaybeUninit`. Although this is unsafe in Rust, we never read the value returned and therefore preserve safety. Because the else branch has an underspecified type (`MaybeUninit` can be implemented for any Rust type), the type inference algorithm will use the unreachable branch to determine the inferred return type. We show an example of the invocation and expanded code in Figure 2.

```
q!(1 + 1) // expands to...
|set_output: &mut String| {
    *set_output = "1 + 1".to_string();
    if false {
        1 + 1 // drives type inference
    } else {
        unsafe {
            MaybeUninit::uninit().assume_init()
        }
    }
} // inferred type: impl Fn(&mut String) -> i32
```

Figure 2. Quoting a simple arithmetic expression, and the expanded result.

The inferred type of the quoted expression is a function that returns a value of the same type as the quoted expression. This is our core mechanism that enables type-safe metaprogramming, but so far this expression does not implement the `Quoted` trait. To do this, we need to provide a mechanism for splicing a quoted expression provided by a closure value. We show this implementation in Figure 3, which uses the syn library to parse the string of tokens. When `splice` is called, it invokes the closure with a mutable string reference. The closure writes the quoted tokens into `output` (e.g., `"1 + 1"`) and returns an uninitialized value of the expression's type into `result`. Because `result` is never read, we use `std::mem::forget` to prevent its destructor from running, and then parse `output` into a token stream.

```
impl <T, F> Quoted<T> for F
where F: FnOnce(&mut String) -> T {
    fn splice(self) -> TokenStream {
        let mut output = String::new();
        let result = self(&mut output);

        // so that the uninit is not consumed
        std::mem::forget(result);

        syn::parse_str::<Expr>(&output)
            .unwrap()
            .into_token_stream()
    }
}
```

Figure 3. The trait implementation for quoted expressions.

Because the expanded closure returns an uninitialized value when invoked, we need to be careful to avoid undefined behavior. When a Rust value goes out of scope, the compiler automatically inserts a call to the `Drop` trait for clean-up. Because `result` is uninitialized, calling `Drop` on it

would be undefined behavior, so we use `std::mem::forget` to suppress the destructor. The remaining logic converts the string in `output` into a token stream using the `syn` library, so that we can manipulate it with rich APIs for Rust ASTs.

3.2 Splicing and Code Generation

Once a piece of code has been quoted, a library author can splice it into generated code using the `splice` method. This method takes a quoted value and produces an untyped token stream capturing the expression. The spliced expression can then be used in any context where a Rust expression is expected, such as in a function body or as an argument to another function.

An important invariant to maintain while splicing is that the spliced code should *always* compile, because the quoted code is already typechecked. Maintaining syntactic correctness is straightforward, since the quoted code already successfully parsed. What is more challenging is ensuring that the spliced code typechecks, since the original typechecking environment may be different than the spliced context.

A particularly tricky case is when the quoted code is for a closure whose input and output types were inferred. When the expression was initially quoted, the type inference algorithm can use constraints on the quoted value to infer types for the closure. But when the quoted code is spliced into a new context, the type inference algorithm may choose different types or fail to infer types altogether. This is a common problem when writing dataflow-oriented libraries with Stageleft, whose APIs often take quoted closures as input.

```
// provided by Stageleft
fn fn1_hint<T, U>(f: impl Fn(T) -> U)
    -> impl Fn(T) -> U {
    f
}

// user code
let quoted: impl Quoted<impl Fn(u32) -> u32> =
    q!(|x| x + 1);

quoted.splice_fn1() // generates the tokens:
"stageleft::fn1_hint::<u32, u32>(|x| x + 1);"
```

Figure 4. Splicing a quoted closure with type hints.

To address this, Stageleft provides mechanisms for splicing a quoted expression *along with* type hints that guide the type inference algorithm. To do this, we rely on Rust's `std::any::type_name` API, which allows us to obtain a string representation of a type. Because Rust specializes each generic function invocation, the string representation will always be a fully expanded type path. The string types can then be parsed back into a Rust AST and included in the tokens generated by the splicing process.

For example, when a library splices an expression using an API such as `splice_fn1` (there are many variants, one for each closure arity), we obtain the input and output types from the `Quoted<T>` trait and include them in the spliced code. When Rust type checks the spliced code, it will use these types as constraints when inferring the type of the closure. We show an example of this in Figure 4, which uses the type hints to preserve inferred types.

4 Scope Hygiene

When quoted code is a pure, local expression without any external references, it is easy to splice it into another context without changing the meaning of the code. When quoted code references external symbols, however, we need to maintain *hygiene* [5] and ensure that the spliced code resolves the same symbols as the original code. An *unhygienic* macro allows the context of the spliced code to affect its meaning, which is the default behavior of Rust procedural macros. Stageleft aims to provide a hygienic quoting mechanism that lets developers use metaprogramming with confidence.

There are two major components to hygiene. The first deals with references to free variables, which are local values that were defined outside the scope of the quoted code; we will discuss this in Section 5. In this section, we focus on hygienic references to types and function symbols that require special care to re-resolve in spliced code.

4.1 Local Symbols

As we saw earlier, when quoting code with `q!`, the provided expression is emitted as-is to be typechecked. This means that quoted code is free to reference a type or function that is available in its scope. When the code is spliced, however, the context may be a different module or crate and the relative paths used in the quoted code may not resolve correctly.

The first step Stageleft takes to address this is to capture metadata about where an expression was quoted, so that we can attempt to replicate the original context at the splice site. Rust helpfully provides a macro called `module_path!` that returns the current module path as a string, which we can store in the quoted value next to the expression AST. Then, when the quoted code is spliced, we can import all symbols from that module so that they are available in the spliced code. We show an example in Figure 5, where the quoted code references a local function `helper`.

This approach has a few limitations due to the Rust compiler. It cannot handle references to functions or types defined *inside* another function, since Rust does not provide a way to directly refer to such a symbol. Currently, Stageleft warns developers to avoid this pattern in documentation, but we hope to provide a more robust solution via custom lints or deeper integration with the Rust compiler itself.

Another challenge is that the quoted code may reference symbols that are not publicly visible from other modules, or

```rust
pub mod foo {
    pub fn helper() -> i32 { 42 }

    fn bar() {
        let q: impl Quoted<i32> = q!(helper());
        // expands to:
        |o: &mut String, m: &mut &'static str| {
            *m = module_path!(); // "crate::foo"
            ...
        };

        q.splice() // generates the tokens:
        "{ use crate::foo::*; { helper() } }"
    }
}
```

Figure 5. An example of quoting an expression that references a local function.

symbols which are imported using a use statement. If the quoted code is spliced into another module, these symbols will not be available in that context. To address this, Stageleft applies a build-time step to generate mirrors of each module in the program that publicly expose all local and imported symbols. Then, instead of importing symbols from the original module, the spliced code imports from the mirror module. We show an example of mirroring in Figure 6, where the quoted code references a symbol that is not publicly visible.

```rust
mod foo {
    use std::cmp::max;
    fn helper() -> i32 { 42 }

    pub fn bar() -> impl Quoted<i32> {
        q!(max(1, helper())).splice()
        // generates the tokens:
        r#"{
            use crate::__staged::foo::*;
            { max(1, helper()) } }
        "#
    }
}

mod __staged { // generated mirror module
    pub mod foo {
        pub use std::cmp::max;
        pub fn helper() -> i32 { 42 }
        pub use crate::foo::bar;
    }
}
```

Figure 6. An example of the mirror module generated to resolve private symbols.

The mirror module simply duplicates every private declaration in each module, and re-exports any symbols that are already public. This means that some functions will be duplicated in the mirror module, but our generation process takes care that the behavior is unchanged. Overall, this approach is a *hack* to work around the privacy restrictions in Rust, but it works well in practice and avoids having to modify the compiler. In future work, these limitations could be addressed by mechanisms to bypass privacy restrictions for generated code or to specify a scope in which symbols should be resolved. These concerns have already been raised in the Rust community for procedural macros; we believe that staging provides a compelling use case for this.

4.2 Resolving Paths

The other set of references that need to be handled with care are *relative paths*, which can use keywords like `self`, `super`, and `crate` to refer to the current module, parent module, or crate root, respectively. If these symbols are passed through directly to the spliced code, they will not resolve correctly in the new context. To address this, Stageleft performs lightweight rewrites over the quoted AST to transform these relative paths into absolute ones.

Conveniently, paths in Rust are syntactically distinguishable from other expressions (unlike languages like Java or Scala), so we can traverse the AST of the quoted expression and rewrite any relative paths based on the known module path. For example, if the quoted code references `self::foo` and the module path is `crate::bar`, we rewrite the path to `crate::bar::foo`. When the quoted code needs to be spliced into a different crate than where it was quoted, we further rewrite the `crate` keyword to the name of the crate where the quoted code was defined.

By appropriately resolving local symbols, developers using libraries powered by Stageleft can easily pass in custom quoted code without worrying about restrictions about what can be quoted. In particular, developers are free to extract common runtime logic into helper functions and define custom types that are used in the quoted code. This makes it much more practical to introduce staging since it has limited effects on how developers write their application logic.

5 Free Variables

So far, we have only considered quoted code with external references to *symbols* such as function declarations and types. But many developers need their quoted code to *capture* local values with semantics similar to how a closure captures locals. This is a complex topic in staging, since quoted code cannot capture arbitrary local values[2] since the local values

[2]Some staging libraries [7, 10] allow capturing values of any type because the generated code is JIT compiled into the existing runtime. Because Rust is AOT compiled, Stageleft code is typically compiled into an independent binary, so we do not have the same flexibility.

will not be available in the spliced context—a completely independent program!

References to locals come in many forms for various purposes. A quoted reference may refer to a constant value that should be inlined into the spliced code, a value that will be materialized elsewhere in the generated program at runtime, or even another quoted expression for composition. All of these cases require different semantics, but they all show up in quoted expressions in the same form: a *free variable*.

Free variables refer to any symbol that appears in a quoted expression but is not defined in the quoted code. Stageleft automatically detects free variables using a lightweight program analysis, and then uses a trait system to handle them appropriately based on their type. Handling free variables requires careful reasoning about Rust's borrow checker, to ensure that resolved references are valid in the spliced code. With free variables backed by traits, Stageleft allows developers to build powerful abstractions around quoting.

5.1 Syntax-Driven Analysis

The first step in handling free variables is to identify them in the quoted code. The core idea is straightforward: we walk the AST to build a set of *bound* variables (those declared within the quoted expression), so that any remaining variable references can be classified as free. Stageleft performs this analysis syntactically, since the `q!` macro runs before typechecking and therefore does not have access to symbol information. To walk the AST, we leverage the `syn` library, which provides convenient visitor patterns for traversing and mutating Rust ASTs in-place. Visitors help ensure coverage of all Rust syntax, and allow us to focus on the relevant parts of the AST.

Our visitor maintains a stack of scopes, which track the set of bound symbols declared in the current block or its parents. When a new block is entered (such as a closure declaration or expression block), we push a new scope onto the stack since any declarations inside will not be visible outside. When a new symbol is introduced through a variable declaration or closure parameter, we add it to the current scope. As we walk the AST recursively, every time we encounter an identifier, we check if it is in any of the scopes on the stack; if it is not, we add it to the set of free variables.

A unique challenge when handling Rust ASTs is that the quoted code may itself use Rust macros. Because macros consume arbitrary tokens, the parsed AST stops at the invocation and does not include the macro expansion. This can result in the visitor missing free variables if the macro is invoked with an expression containing variable references. To tackle this, Stageleft attempts to parse each argument to the macro as a Rust expression, and if it succeeds we then walk the AST of the argument as usual. While this is not a perfect solution (in particular, it fails to handle formatting macros that use string interpolation syntax), it works well for common cases like `dbg!`. There have been proposals to allow Rust macros to change expansion order, which would allow us to handle this robustly, but they have not yet been accepted into the language.

5.2 Splicing Free Variables

Once we have identified the free variables, the next goal is to determine how each one should be represented in the spliced code. Different free variables require different treatment: a constant integer should be inlined as a literal, a quoted expression should be spliced as its tokens, and a runtime variable should emit a reference by name. To support this, Stageleft uses a trait called `FreeVariable`, which is implemented by all types that can appear as free variables in quoted code.

There are two responsibilities of the `FreeVariable` trait. The first is to provide a method for splicing the free variable into the quoted code, which is done in `to_tokens`. This method consumes the free variable as an owned value and returns a token stream that represents the free variable. The token stream will be used to compute the value of the free variable inside the quoted code. Because we take ownership of the free variable container, we can enforce borrow checking rules by preventing illegal aliasing.

The second is to provide the type of the expanded variable, which is done by using a *generic type* in the trait definition. By having the trait provide the type the free variable will expand to, we can ensure that the quoted expression is type checked with respect to the spliced value, rather than its container. The trait also provides an `uninitialized` method for obtaining a dummy value for typechecking purposes. We show the definition of the `FreeVariable` trait in Figure 7.

```
pub trait FreeVariable<O> {
    fn to_tokens(self) -> TokenStream;
    fn uninitialized(&self) -> O {
        unsafe {
            MaybeUninit::uninit().assume_init()
        }
    }
}
```

Figure 7. The core definition of the `FreeVariable` trait.

When a quoted expression references a free variable, the `q!` macro generates calls to the `FreeVariable` implementation for each referenced variable. First, we obtain dummy values for each free variable, which are stored in variables with the same identifier as the free variable (which shadows them for use in typechecking). Then, we generate the quoted tokens by interpolating the free variable tokens into the rest of the expression.

Notably, we *do not* insert the free variable tokens directly at the point where the variable was referenced. This is because the same free variable may be referenced multiple

```
q!(x + 1) // expands to:
|set_output: &mut String, ...| {
    let x_value = FreeVariable::uninitialized(&x);
    let x_tokens = FreeVariable::to_tokens(x);
    *set_output = quote! {
        let x = #x_tokens;
        { x + 1 }
    }.to_string(); // "let x = ...; { x + 1 }"

    if false {
        let x = x_value;
        { x + 1 }
    } else { ... }
}
```

Figure 8. An example of quoting an expression that references a free variable, expanded using traits.

times in the quoted code, but to preserve semantics we cannot inline the free variable tokens twice as they may involve mutable references. Instead, we compute the value of the free variable at the top of the quoted expression, and then store the original tokens as-is. We show an example of this expansion in Figure 8, where the quoted code references a free variable `x`. The expanded code uses the `quote!` macro from syn, which provides a simple way to splice tokens.

5.3 Composing Quoted Values

Quoted expressions, which implement the `Quoted` trait, also implement `FreeVariable`! This makes it possible to compose quoted code together into complex expressions and thus enables type-safe code generation.

A quoted expression of type `impl Quoted<T>` implements `FreeVariable<T>`, which means that the quoted expression can be spliced as a value of type `T`. To generate the spliced tokens, we simply pass through the tokens of the quoted expression as-is. This is safe from a hygiene perspective because we guarantee that all free variables have been resolved before the quoted expression is spliced, so no symbols outside the spliced tokens will be used. Furthermore, because splicing an expression consumes it by-value, we preserve Rust's ownership semantics.

To see the free variable mechanism in action, let us walk through an end-to-end example composing two quoted snippets. Both perform a simple arithmetic computation, but the second relies on the value computed by the first. Because splicing a free variable consumes it by-value, we can no longer use it after it is captured. We show this composition in Figure 9.

While Rust traits are typically handled using static dispatch by specializing to the trait implementation, Rust also offers *trait objects*, which use dynamic dispatch to allow for dynamic fulfillment of a trait. We can use this in Stageleft to

```
let quoted_a = q!(1 + 1);
let quoted_b = q!(quoted_a * 2);

quoted_b.splice() // generates the tokens:
r#"{
    let quoted_a = { 1 + 1 };
    { quoted_a * 2 }
}"#
```

Figure 9. An example of composing quoted expressions.

perform dynamic code generation, by passing around quoted trait objects across traditional control flow mechanisms.

This allows us to achieve a common use case for staged programming: partial evaluation. In situations where some inputs to a computation are known at stage-time, we can perform code generation that bakes in that input for higher performance. A classic example of this is raising an unknown base to a known power. Instead of relying on expensive instructions or runtime control-flow, we can instead generate an expression that uses a constant number of multiplications. By using staged programming, we can easily implement this optimization in a *type-safe manner*, guaranteeing that the expanded code will always compile.

The `Quoted` trait provides a helpful `boxed` API that turns a quoted expression into a heap-allocated trait object. This allows different branches in the recursion to return different quoted values, while the function still returns a single unified type. We also require the `base` to implement `Copy`, allowing it to be spliced several times.

We show the full implementation and an example of expansion in Figure 10, where the expanded code more efficiently computes the power by breaking it down into squaring multiplications. Each intermediate value of `v` is computed by splicing the nested quoted expression, so the function recursively builds up the result. The boxing and dynamic dispatch in staged programs is a *zero-cost abstraction*—the spliced code is not affected by the indirection and has the same raw performance.

5.4 Custom Free Variables and Ownership

In addition to composing quoted expressions, Stageleft also allows library authors to define their own container types that can be spliced into quoted code. These can be used to interact with the borrow checker in more complex ways by permitting aliased references to shared data or to enable access to data outside the scope of the spliced code.

A common special free variable type is for constant data available at stage-time that is referenced from quoted code. For example, if we have a variable storing an integer, we may want to reference it in a quoted expression with the integer value inlined as a constant literal. To do this, we need to implement the `FreeVariable` trait for primitive types such as integers, strings, and other data types which can

```
fn exp(
    base: impl FreeVariable<i32> + Copy,
    power: u32
) -> impl Quoted<i32> {
    if power == 1 {
        q!(base).boxed()
    } else if power % 2 == 0 {
        let v = exp(base, power / 2);
        q!(v * v).boxed()
    } else {
        let v = exp(base, power / 2);
        q!((v * v) * base).boxed()
    }
}

exp(2, 5).splice() // expands to (simplified):
{
    let base = 2;
    let v = {
        let v = {
            let base = 2;
            { base }
        };
        { v * v } // 2^2
    };
    { (v * v) * base } // 2^5
}
```

Figure 10. An example of partial evaluation for raising an unknown base to a fixed power.

be transferred into the quoted code as a literal. We show examples of these trait implementations in Figure 11, where we implement the trait for integers and strings.

```
impl FreeVariable<i32> for i32 {
    fn to_tokens(self) -> TokenStream {
        syn::parse_str::<Expr>(
            &self.to_string()
        ).unwrap().into_token_stream()
    }
}

impl FreeVariable<&'static str> for &str {
    fn to_tokens(self) -> TokenStream {
        syn::parse_str::<Expr>(
            &format!("\"{}\"", self)
        ).unwrap().into_token_stream()
    }
}
```

Figure 11. Implementations of the `FreeVariable` trait for integers and strings.

Because the container type is a primitive Rust value, such free variables also implement the `Copy` trait and can be used in multiple quoted expressions, because their spliced code is guaranteed to be pure and reference-free. We used this in Figure 10 to allow the base to be passed in as a regular Rust integer, which is then spliced into the quoted code as a constant literal.

Another custom free variable type provided by Stageleft is an opt-in escape hatch that allows access to an identifier declared outside of the quoted code. This is useful for referencing inputs to the program that will only be available at runtime, or for accessing global metadata in the generated program. To expose such a variable, Stageleft provides a `RuntimeData<T>` struct that captures a reference to an externally-declared variable of type `T`. The tokens produced by `RuntimeData` appear only in the free variable bindings at the top of the expanded code, not in the quoted body itself, so the hygiene of the rest of the quoted expression is unaffected. This container simply stores the name of the variable under the hood and emits it when spliced.

A key component of Rust's borrow checker is setting restrictions on which types of variables can be referenced in multiple places, versus which can only be consumed once. An integer variable can be referenced many times, while a heap-allocated `Box` will be *moved* when it is used. In Rust, these rules are captured by the `Copy` trait, which is implemented for types that can be transparently copied by value. In Stageleft, we lift these rules to the `RuntimeData` type, which implements `Clone` if the underlying type `T` implements `Copy`. This allows developers to explicitly create multiple references to the same variable, while ensuring that the ownership rules are respected. We show the definition and traits for `RuntimeData` type in Figure 12.

```
struct RuntimeData<T> {
    name: String,
}

impl<T> FreeVariable<T> for RuntimeData<T> {
    fn to_tokens(self) -> TokenStream {
        syn::parse_str::<Expr>(
            &self.name
        ).unwrap().into_token_stream()
    }
}

impl<T: Copy> Clone for RuntimeData<T> {
    fn clone(&self) -> Self {
        RuntimeData { name: self.name.clone() }
    }
}
```

Figure 12. The definition of the `RuntimeData` type and its trait implementations that enforce borrowing semantics.

Stageleft enforces the Rust borrowing rules through trait bounds on the implementations of `FreeVariable`. For example, a `RuntimeData` value can only be duplicated with `Clone` if the underlying type implements `Copy`. A `RuntimeData` value storing a shared pointer (`&T`) can therefore be duplicated into aliased references. On the other hand, variables storing unique pointers (`&mut T`) can only be referenced once. While this is an overly restrictive condition, since mutable pointers can be used in multiple places as long as only one instance is active at a time, we cannot enforce such restrictions in staging and take a more conservative approach.

Currently, instances of `RuntimeData` are manually created by mechanisms *outside* the quoted code and therefore Stageleft cannot guarantee that the variable will typecheck appropriately in the spliced code. In our quoting implementation, variable declarations inside quoted code cannot be referenced in nested quoted expressions, because those nested values must be materialized *before* they are spliced. In future work, we hope to support quoted expressions that explicitly reference free variables that will be declared in parent code, by automatically generating `RuntimeData` instances for these "parent variables".

5.5 Guarantee Boundaries

Stageleft aims to ensure that if staged code compiles, the generated code will also compile. Because Stageleft is a library rather than a compiler extension, this guarantee holds relative to a set of trusted components. We consolidate the relevant cases here.

As shown in Figure 8, the expanded form of a quoted expression has two distinct regions: the free variable bindings at the top (the `let x = #x_tokens;` lines) and the original quoted body below. All references within the quoted body are hygienic: the scope analysis (Section 4) rewrites paths so that symbols resolve to the same declarations regardless of where the code is spliced. The free variable bindings, however, contain tokens produced by each variable's `FreeVariable` implementation, and their correctness depends on that implementation. Stageleft provides core implementations for primitives, quoted expressions, and `RuntimeData`, which cover most use cases. Library authors may add custom implementations for their own types; these are trusted by Stageleft in the same way that any trait implementation is trusted by the code that calls it.

There are several cases where the compile-time guarantee does not hold, each arising from the library-based design:

- **Nested macros in quoted code.** When quoted code invokes a Rust macro, Stageleft cannot inspect the macro's expansion and may miss free variables inside it. Stageleft partially mitigates this by attempting to parse macro arguments as expressions, but this heuristic does not cover all cases (e.g., formatting macros with string interpolation).
- **Inner function references.** Rust does not provide a way to name a function defined inside another function from an external module. If quoted code references such a local function, the mirror module cannot re-export it and the spliced code will fail to compile.
- **RuntimeData scoping.** `RuntimeData` instances are constructed manually, so Stageleft cannot verify that the named variable will exist or have the correct type at the splice site. This is an intentional escape hatch for spliced code that is wrapped with additional context.
- **Closure type inference.** When quoted closures rely on type inference, the inferred types may fail to resolve at the splice site. Stageleft mitigates this with type hints (Figure 4), but edge cases remain when the type hint mechanism cannot fully reconstruct the original inference context.

These limitations are inherent to operating within Rust's existing macro infrastructure. In each case, the failure mode is a compile-time error in the generated code rather than a silent runtime bug. We are actively working with the Rust community to explore language changes that would address these issues, such as a serializable format for types that would strengthen inference guarantees and mechanisms to bypass privacy restrictions for generated code.

6 Expansion and Entrypoints

The last piece of Stageleft is to provide a way to compile staged code into runnable binaries. To give library authors flexibility in code generation, Stageleft provides two ways to expose staged snippets: as a macro or as an independent Rust source file. The former is useful for developers who want to consume staged logic as an API in other Rust code, while the latter is useful when staged code needs to be wrapped with additional runtime logic and launched in a particular way.

6.1 Type-Safe Rust Macros

Staged logic written by Stageleft can be wrapped in a *macro entrypoint*, which is accessible from other Rust code as a *standard proc-macro*. The broad approach of exposing staging through macros is a standard technique used in languages like Scala [12]; it enables a high-level API for writing macros that are *guaranteed* to typecheck when expanded.

Developers can create a macro entrypoint by annotating a regular Rust function with `#[stageleft::entry]`, which is itself a Rust macro. The function must have a return type of the form `impl Quoted<T>`; this quoted expression will be emitted as part of the macro expansion. The `stageleft::entry` macro wraps the function into a procedural macro definition that can be used in other Rust crates.

The parameters to the entrypoint function define inputs to the macro. Stageleft entrypoints cannot consume arbitrary tokens as inputs, as all inputs must be well-formed Rust expressions in order for them to be exposed as type-safe staged

values. When the macro is invoked, the input tokens are parsed into a list of Rust expression ASTs and then handled according to the corresponding type in the parameter list.

If the parameter is a `RuntimeData` type, the corresponding argument in the invocation will be evaluated and stored into a variable, and then handed off for opaque use by the staged logic. Like with free variables, the expression *will not* be inlined into the expanded code in order to preserve typical Rust evaluation semantics where function arguments are evaluated eagerly. This is useful for computational inputs to the staged code which will be computed at runtime.

If the parameter is any other type, it must implement a `ParseLiteral` trait defined by Stageleft. Each implementation of this trait defines logic to extract a value of type `T` given tokens representing a literal of that type (this is a dual of our previous `FreeVariable` implementations). Parsing literals is useful for cases where an integer or string will be passed to the macro as an inline literal, and allows the macro to generate code based on that input.

In order to guarantee type safety in the macro expansion, the quoted expression is wrapped into a function definition that takes in all the `RuntimeData` inputs as parameters. This ensures that if an expression passed into the macro invocation has the wrong type, the error will be isolated to that parameter instead of causing typechecking failures in the quoted code. We show an example of wrapping the exponentiation function in a macro entrypoint in Figure 13, where the macro takes in a base and power as parameters.

Staged macros are an effective way to incrementally introduce staged programming to an existing Rust codebase. Because macro entrypoints use existing Rust mechanisms for code generation, they are easy to use and have strong support from tooling like IDEs—even features such as code completion work on arguments passed to the macro. Furthermore, the macro entrypoint hides the implementation details from the end user, who does not need to know that staging is used under the hood. While not all macros can be written using staged programming, Stageleft is a great way to improve the correctness of macros that fit the staged model.

```rust
#[stageleft::entry]
pub fn exp(
    base: RuntimeData<i32>, power: u32
) -> impl Quoted<i32> { ... }

// generated macro entrypoint:
#[proc_macro]
pub fn exp(input: TokenStream) -> TokenStream {
    let input_parsed: Vec<syn::Expr> = ...;
    let output_core = Quoted::splice(exp(
        RuntimeData {
            name: "value".to_string()
        },
        <u32 as ParseLiteral>::parse(
            &input_parsed[1],
        )
    ));

    quote! {
        fn expand_staged(base: i32) -> i32 {
            #output_core
        }
        expand_staged(#(&input_parsed[0]))
    }
}

// invocation (in another crate):
let dynamic_input = ...;
let result = exp!(dynamic_input, 5);
```

Figure 13. An example of a Stageleft macro entrypoint.

6.2 Independent Crates

For staged libraries that need to control how the staged code is compiled and run, Stageleft also provides a way to generate independent Rust binaries from staged code. While this approach requires more manual intervention from the library author, it allows the library to control the end-to-end staging process rather than relying on the developer to invoke a staged macro.

Our approach to independent compilation is based on the trybuild [20] library, which was originally intended to provide a way for Rust authors to write unit tests for the typechecking behavior of their libraries. Under the hood, trybuild generates an independent Rust crate containing the code to be tested, and then runs the Rust compiler on that crate to check for type errors. The trybuild library handles key components that we need in staging, such as efficient re-use of the existing compiler cache and propagating Rust compiler flags. Stageleft uses a fork of trybuild that preserves many of its core compilation features but allows library authors to plug in their own code generation logic.

To splice staged code into a new crate, library authors can use the existing `splice` API to obtain the tokens for a quoted expression. Then, they can wrap the generated code into a generated entrypoint file such as a `main` function. This is then handed off to the trybuild library, which generates a new Rust crate with the provided code and compiler configuration derived from the original crate that is using the library.

Once the Rust crate is generated, the library can programmatically invoke the Cargo build tool to compile the staged code into a Rust binary, and can even invoke it. Although Stageleft does not yet include mechanisms to dynamically load the compiled code into the existing process, we believe that this is a promising direction to bring Stageleft closer in capability to staging libraries for other languages.

7 Evaluation

Stageleft is used extensively in the Hydro distributed programming framework [6]. In this section, we describe how Hydro relies on Stageleft and present a benchmark measuring the performance impact of staged compilation.

7.1 Case Study: Hydro

Hydro is a framework for writing distributed programs in a single unified codebase. Developers write dataflow pipelines over streams that are parameterized by a *location type* indicating which machine the data resides on (e.g., `Process<P1>` or `Cluster<C1>`). All operator closures (map, filter, fold, etc.) are wrapped in Stageleft's `q!` macro, so that the framework captures their ASTs.

When a Hydro program is compiled, the framework collects an intermediate representation (IR) of the full dataflow graph. Each IR node stores the quoted closure as an AST. At staging time, the framework partitions this IR by location and emits a separate Rust binary for each machine using the independent crate mechanism (Section 6.2). Staging is core to Hydro, as it allows developers to write distributed logic with types that propagate across locations.

```
fn my_pipeline(
    workers: &Cluster<Workers>,
    leader: &Process<Leader>,
    threshold: RuntimeData<f64>,
) {
    let on_worker = workers
        .source_iter(q!(read_data()))
        .map(q!(|raw| {
            // local module path
            self::parse_record(raw)
        }));

    on_worker
        .send(leader) // network boundary
        .map(q!(|(_, v)| v))
        .filter(q!(|r| {
            // free variable reference
            r.score > threshold
        }))
        .fold(
            q!(|| 0.0),
            q!(|acc, r| *acc += r.score),
        )
        .for_each(q!(|total|
            println!("total: {total}")
        ));
}
```

Figure 14. A simplified Hydro program. Closures reference a user-defined helper (`parse_record`), a runtime variable (`threshold`), and rely on type inference hints.

To illustrate how Stageleft's features are exercised in practice, we walk through a simplified Hydro program in Figure 14. A cluster of workers produces data that is sent to a leader, which filters and aggregates it. The closures passed to `map`, `filter`, and `fold` are all quoted with `q!`, so Hydro captures their ASTs. The hygiene mechanism (Section 4) rewrites the path to `parse_record` so that the spliced code resolves the same symbol. The closure passed to `filter` references `threshold`, a `RuntimeData` value (Figure 12) that will be resolved in the generated code. Finally, each closure relies on type inference; Hydro uses the type-hint mechanism (Figure 4) to preserve the inferred types when the closures are spliced.

Hydro partitions the IR at the `send` boundary and emits two Rust binaries. The worker binary contains the `read_data`, `parse_record`, and serialization logic. The leader binary contains the deserialization, filter, fold, and print logic. Within each binary, the chain of operators is fused into a tight loop because Stageleft emits closures as concrete Rust code rather than trait objects, allowing the Rust compiler to inline across operator boundaries.

7.2 Stream Fusion Microbenchmark

As shown in the case study, Hydro's staged compilation model emits each operator closure as concrete Rust code, which allows the Rust compiler to inline across operator boundaries and fuse chains of transformations into tight loops. A framework that builds pipelines dynamically, storing closures as trait objects behind `Box<dyn FnMut>`, loses this ability because each operator boundary becomes a virtual function call that the compiler cannot see through.

To quantify this difference, we benchmark a pipeline of 10 map operations and 5 filter operations applied to one million 64-bit integers using the Criterion benchmarking framework [3]. We compare three configurations:

- **Staged (Hydro)**: A Hydro pipeline where each operator closure is quoted with `q!` and compiled by Stageleft into concrete Rust code. The benchmark drives the generated DFIR dataflow graph synchronously.
- **Dynamic dispatch**: The same operations stored as `Box<dyn FnMut>` trait objects and dispatched through a vector at runtime, simulating a library that builds pipelines without staging. The compiler cannot inline across these dispatch boundaries.
- **Hand-written**: A manually fused single loop that serves as a performance ceiling.

We show a simplified excerpt of the Hydro pipeline definition in Figure 15. Each closure is quoted with Stageleft's `q!` macro. At staging time, the Hydro compiler splices these closures into a DFIR dataflow graph where each operator has a concrete closure type.

```rust
fn pipeline<'a>(s: Stream<u64, Process<'a>>) {
    s.map(q!(|x| x + 1))
     .filter(q!(|x| x % 3 != 0))
     .map(q!(|x| x + 1))
     .filter(q!(|x| x % 5 != 0))
     // ... 10 more operators
     .embedded_output("output");
}
```

Figure 15. The benchmark pipeline with quoted closures.

Configuration	Time (95% CI, ms)	Relative
Hand-written	0.910–0.917	1.00×
Staged (Hydro)	2.335–2.347	2.56×
Dynamic dispatch	14.24–14.33	15.6×

Table 1. Stream fusion benchmark results (1M elements, 15 operators). Lower is better.

The staged Hydro is over 6× faster than dynamic dispatch, demonstrating that Stageleft's code generation enables the Rust compiler to inline across operator boundaries. The remaining gap between staged compilation and the hand-written loop is due to the DFIR runtime overhead (dataflow scheduling and output collection). In a production Hydro deployment, this overhead is negligible relative to the cost of serialization and communication.

8 Related Work

Staged programming is a well-studied topic with implementations across many languages. Stageleft takes heavy inspiration from these implementations and mirrors many of their decisions, but focuses on supporting Rust without requiring changes to the compiler. In this section, we review some of the most relevant work in this space.

8.1 Staging-as-a-Library

While staging is now built into many modern programming languages, early implementations were often built as libraries on top of existing languages using their metaprogramming features. A prominent example is Lightweight Modular Staging (LMS) [10], a Scala library that adds staging constructs without requiring special compiler support. LMS has been applied to many similar domains as Stageleft, including query processing [11] and streaming frameworks [4].

LMS uses Scala's macro system to provide a high-level API for quoting and splicing code, and uses the Scala compiler API to execute the generated code. Stageleft borrows much of its design from LMS. Stageleft uses the `Quoted` type to capture expressions similar to the `Rep` type in LMS and inserts let bindings for references to free variables like LMS does when interpolating quoted code. Because Scala offers implicit conversions, LMS can *automatically* quote snippets of Scala code without the developer having to explicitly annotate them. Because Rust does not have a similar feature, Stageleft requires quoted code to be wrapped in the `q!` macro.

A common criticism of the LMS approach is that it incurs a latency penalty when compiling the generated code. When using Stageleft to generate an independent crate, we face a similar limitation as the Rust compiler must process the generated program. Recent work has focused on unifying macros and staging [12], which allows the staged code to be expanded at compile-time. Stageleft's macro entrypoints feature enables a similar approach, but requires more manual separation of the staged code from the rest of the program. With support for dependent types on the horizon for Rust, it may soon be possible to further unify macros and staging in Rust by tracking staged snippets in the type system [7].

8.2 Metaprogramming in Rust

Today, Rust offers two vastly different macro systems: declarative macros and procedural macros. Declarative macros can be defined in the same crate as the code that uses them, but are limited to a simple pattern matching language. Procedural macros require low-level token manipulation and are defined in a separate crate, but allow for arbitrary code generation logic. Rust developers today face a challenging tradeoff between the two systems, which have vastly different semantics and limitations. Staging offers an opportunity to unify these systems by providing a mechanism that is higher-level than procedural macros but richer than declarative macros.

A key area of focus has been the *security* of macro systems in Rust [18]. Procedural macros in Rust can contain arbitrary Rust code, including logic that reads files and launches other programs. With growing concern about supply-chain attacks, the Rust community has been exploring ways to prevent rogue macro logic from exploiting development machines. A promising direction is to restrict procedural macros to *const functions* [17], a special function type that restricts the internal logic to be deterministic and have no side effects. Const functions are interpreted at compile-time, making them an excellent candidate for lifting staging into a first-class language feature that does not require the macro indirection.

9 Conclusion

In this paper, we presented Stageleft, a Rust library for staged programming that allows libraries to leverage code generation logic in a type-safe manner. Stageleft is designed to be easy to use and integrate into Rust libraries, while providing strong guarantees about the hygiene of the generated code. Stageleft leverages traits in Rust to provide extensible support for free variables without requiring support from the compiler. We showed how programs written in Stageleft can be exposed as macros or independent Rust binaries, enabling use across domains such as distributed systems to enable new developer interfaces and performance optimizations.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265˘283.

[2] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. https://mlsys.org/Conferences/doc/2018/146.pdf

[3] Brook Heisler. 2025. Criterion.rs: Statistics-driven Microbenchmarking in Rust. https://github.com/bheisler/criterion.rs

[4] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 285˘299. https://doi.org/10.1145/3009837.3009880

[5] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 151˘161. https://doi.org/10.1145/319838.319859

[6] Shadaj Laddad. 2025. *Programming Models for Correct and Modular Distributed Systems*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-85.html

[7] Shadaj Laddad and Koushik Sen. 2020. Fluid quotes: metaprogramming across abstraction boundaries with dependent types. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Virtual, USA) *(GPCE 2020)*. Association for Computing Machinery, New York, NY, USA, 98˘110. https://doi.org/10.1145/3425898.3426953

[8] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (may 2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

[10] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. Association for Computing Machinery, New York, NY, USA, 127˘136. https://doi.org/10.1145/1868294.1868314

[11] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1, Article 4 (April 2018), 45 pages. https://doi.org/10.1145/3183653

[12] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE 2018)*. Association for Computing Machinery, New York, NY, USA, 14˘27. https://doi.org/10.1145/3278122.3278139

[13] Nicolas Alexander Stucki. 2023. *Scalable Metaprogramming in Scala 3*. Ph. D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-8257

[14] Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[15] Walid Taha and Tim Sheard. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. Association for Computing Machinery, New York, NY, USA, 203˘217. https://doi.org/10.1145/258993.259019

[16] Walid Mohamed Taha. 1999. *Multistage programming: its theory and applications*. Oregon Graduate Institute of Science and Technology.

[17] The Rust Team. 2025. *Rust Reference: Constant evaluation*. https://doc.rust-lang.org/reference/const_eval.html

[18] David Tolnay. 2025. *Pre-RFC: Sandboxed, deterministic, reproducible, efficient Wasm compilation of proc macros*. https://internals.rust-lang.org/t/pre-rfc-sandboxed-deterministic-reproducible-efficient-wasm-compilation-of-proc-macros/19359

[19] David Tolnay. 2025. *Syn: Parser for Rust source code*. https://github.com/dtolnay/syn

[20] David Tolnay. 2025. *Trybuild: Test harness for ui tests of compiler diagnostics*. https://github.com/dtolnay/trybuild