Polymorphisation: Improving Rust compilation times through intelligent monomorphisation

David Wood (2198230W)

April 15, 2020

MSci Software Engineering with Work Placement (40 Credits)

ABSTRACT

Rust is a new systems programming language, designed to provide memory safety while maintaining high performance. One of the major priorities for the Rust compiler team is to improve compilation times, which is regularly requested by users of the language. Rust's high compilation times are a result of many expensive compile-time analyses, a compilation model with large compilation units (thus requiring incremental compilation techniques), use of LLVM to generate machine code, and the language's monomorphisation strategy. This paper presents a code size optimisation, implemented in rustc, to reduce compilation times by intelligently reducing monomorphisation through "polymorphisation" - identifying where functions could remain polymorphic and producing fewer monomorphisations. By reducing the quantity of LLVM IR generated, and thus reducing code generation time spent in LLVM by the compiler, this project achieves 3-5% compilation time improvements in benchmarks with heavy use of closures in highly-generic, frequently-instantiated functions.

1. INTRODUCTION

Rust is a multi-paradigm systems programming language, designed to be syntactically similar to C++ but with an explicit goal of providing memory safety and safe concurrency while maintaining high performance. The first stable release of the language was in 2015 and each year since 2016, Rust has been the "most loved programming language" in Stack Overflow's Developer Survey [11, 12, 13, 14]. In the years since, Rust has been used by many well-known companies, such as Atlassian, Braintree, Canonical, Chef, Cloudflare, Coursera, Deliveroo, Dropbox, Google and npm.

Each year, the Rust project runs a "State of Rust" survey, asking users of the language on their opinions to help establishment development priorities for the project. Since 2017 [22, 19, 20], faster compilation times have consistently been an area where users of the language say Rust needs to improve.

There are a handful of reasons why rustc, the Rust compiler, is slow. Compared to other systems programming languages, Rust has a moderately-complex type system and has to enforce the constraints that make Rust safe at compilation time - analyses which take more time than those required for a language with a simpler type system. Lots of effort is being invested in optimising rustc's analyses to improve performance.

In addition, Rust's compilation model is different from other programming languages. In C++, the compilation unit is a single file, whereas Rust's compilation unit is a crate[1]. While compilation times of entire C++ and entire Rust projects are generally comparable, modification of a single C++ file requires far less recompilation than modification of a single Rust file. In recent years, implementation of incremental compilation into rustc has improved recompilation times after small modifications to code.

Rust generates machine code using LLVM, a collection of modular and reusable compiler and toolchain technologies. At LLVM's core is a language-independent optimiser, and code-generation support for multiple processors. Languages including Rust, Swift, Julia and C/C++ have compilers using LLVM as a "backend". By de-duplicating the effort of efficient code generation, LLVM enables compiler engineers to focus on implementing the parts of their compiler that are specific to the language (in the "frontend" of the compiler). Compiler frontends are expected to transform their internal representation of the user's source code into LLVM IR, an intermediate representation from LLVM which enables optimisation and code-generation to be written without knowledge of the source language.

While LLVM enables Rust to have world-class runtime performance, LLVM is a large framework which is not focused on compile-time performance. This is exacerbated by technical debt in rustc which results in the generation of low-quality LLVM IR. Recent work on MIR[2] optimisations have improved the quality of rustc's generated LLVM IR, and reduced the work required by LLVM to optimise and produce machine code.

Finally, rustc monomorphises generic functions. Monomorphisation is a strategy for compiling generic code, which duplicates the definition of generic functions for every instantiation, producing significantly more generated code than other translation strategies. As a result, LLVM has to perform significant work to eliminate or optimise redundant IR.

This project improves the compilation times of Rust projects by implementing a code-size optimisation in rustc to eliminate redundant monomorphisation, which will reduce the quantity of generated LLVM IR and eliminate some of the work previously required of LLVM and thus improve compilation times. In particular, the analysis implemented by this

[1]Crates are entire Rust projects, including every Rust file in the project

[2]MIR is rustc's "middle intermediate representation" and is introduced in Section 3.1.3

project will detect where generic parameters to functions, closures and generators are unused and fewer monomorphisations could be generated as a result.

Detection of unused generic parameters, while relatively simple, was deliberately chosen, as the primary contribution of this work is the infrastructure within rustc to support polymorphic MIR during code generation, which will enable more complex extensions to the initial analysis (see further discussion in Section 5.1). Despite this, there is ample evidence in the Rust ecosystem, predating this project, that detection of unused generic parameters could yield significant compilation time improvements.

Within Rayon, a data-parallelism library, a pull request [17] was submitted to reduce the usage of closures by moving them into nested functions, which do not inherit generic parameters (and thus would have fewer unnecessary monomorphisations). Similarly, in rustc, the same author submitted a pull request [16] to apply the same optimisation manually to the language's standard library.

Likewise, in Serde, a popular serialisation library, it was observed [21] that a significant proportion of LLVM IR generated was the result of a tiny closure within a generic function (which inherited and did not use generic parameters of the parent function, resulting in unnecessary monomorphisations). In this particular case, the closure contributed more LLVM IR than all but five significantly larger functions.

This project makes all of these changes to rustc, Rayon and Serde unnecessary, as the compiler will apply this optimisation automatically.

In Section 2, this paper reviews the existing research on optimisations which aim to reduce compilation times or code size through monomorphisation strategies. Next, Section 3 describes the polymorphisation analysis implemented by this project in more detail and Section 4 presents a thorough analysis of the compilation time impact of polymorphisation implemented in rustc. Finally, Section 5 summarises the result of this research and describes future work that it enables.

2. BACKGROUND

2.0.1 *Monomorphisation-related Optimisations*

Optimisation of monomorphisation to improve compilation time is a subject that has received little attention. However, within the Java ecosystem, lots of literature exists on addressing runtime performance impacts that result from monomorphisation (or a lack thereof).

Ureche et al. [24] present a similar optimisation, "Miniboxing", designed to target the Java Virtual Machine and implemented in Scala. Homogeneous and heterogeneous approaches to generic code compilation are presented. Heterogeneous approaches duplicate and adapt code for each type individually, increasing code size (also known as monomorphisation, the approach taken in Rust and C++). Homogeneous translation generates a single function but requires data to have a common representation, irrespective of type, which is typically achieved through pointers to heap-allocated objects. Ureche et al. identify that larger value types (e.g. integers) can contain smaller value types (e.g bytes) and that this can be exploited to reduce the duplication necessary in homogeneous translations. When implemented in the Scala compiler, performance matched heterogeneous translation and obtained speedups of over 22x compared to homogeneous translation, with only modest increases in code size.

Dragos et al. [3] contribute an approach for compilation of polymorphic code which allows the user to decide which code should be specialised (monomorphised). This approach allows for specialised code to be compiled separately and mixed with generic code. Their approach, again implemented in Scala, introduces an annotation which can be applied to type parameters. This annotation instructs the compiler to specialise code for that type parameter. Generic class definitions require that the approach presented must be able to specialise entire class definitions while maintaining subclass relationships so specialised and generic code can interoperate. Evaluation is performed on the Scala standard library and a series of benchmarks where runtime performance is improved by over 20x for code size increases of between 16% and 161%, however this approach requires programmers correctly identify functions to annotate.

Stucki et al. [18] identify that specialisation of generic code, which can improve performance by an order of magnitude, is only effective when called from monomorphic sites or other specialised code. Performance of specialised code within these "islands" regresses to that of generic code when invoked from generic code. Stucki et al's implementation provides a Scala macro which creates a "specialised body" containing the original generic code, where type parameters have Scala's specialisation annotation. Dispatch code is then added, which matches on the incoming type and invokes the correct specialisation. While this approach induces runtime overhead, this is made up for in the performance improvements achieved through specialisation. Their approach is implementable in a library without compiler modifications and achieves speedups of up to 30x, averaging 12x when benchmarked with ScalaMeter.

Ureche et al. [23] discuss efforts by the Java platform architects in Project Valhalla to update the Java Virtual Machine (JVM)'s bytecode format to allow load-time class specialisation, which will enable opt-in specialisation in Java. Java is contrasted to Scala, which has three generic compilation schemes (erasure, specialisation and Ureche et al's miniboxing). The interaction of these schemes can cause subtle performance issues which also affect Project Valhalla. In particular, miniboxed functions are still erased when invoked from erased functions, and boxing is required at generic compilation scheme boundaries. This paper presents three techniques to eliminate the slowdowns associated with miniboxing. The authors validate their technique in the miniboxing plugin for Scala, producing performance improvements of 2-4x.

These papers focus on languages based on the JVM due to the performance impact that using primitives in generic code can have. In order to preserve bytecode backwards compatibility, the JVM requires that value types be converted to heap objects, typically through boxing, when interacting with generic code. Scala has been of particular focus in this research because of the inclusion of specialisation annotations as a language feature.

2.0.2 *Code Size Optimisations*

In order to improve Rust's compilation times, this project implements an optimisation to reduce the quantity of generated LLVM IR. It is therefore relevant to review research into optimisations which aim to reduce code-size.

Edler von Koch et al. [4] discuss a compiler optimisation technique for reducing code size by exploiting the structural similarity of functions. Structurally similar functions have equivalent signatures and equivalent control flow graphs. Edler von Koch et al. introduce a technique for merging two structurally similar functions by combining the bodies of both functions and inserting control flow constructs to select the correct instructions where the functions differ. The original functions are replaced with stubs which then call the merged function with an additional argument to select the correct code path. The authors implemented this optimisation in LLVM and achieved code size reductions of an average 4% in Spec CPU2006 benchmarks.

Debray et al. [2] present a series of transformations which can be performed by the compiler to reduce code size. Transformations are split into two categories, classical analyses and optimisations and techniques for code factoring. Most relevant to the techniques employed by this project is redundant-code elimination. Redundant computations exist where the same computation has been completed previously and that result is guaranteed to still be available. In their implementation, redundant code elimination techniques were particularly effective at removing redundant computation of global pointer values (on the Alpha processor where the techniques were implemented) when entering functions. Debray et al. evaluate the optimisations by implementing them in a post-link-time optimiser, alto, and comparing the code size of a set of benchmarks before-and-after optimisation by alto, achieving over 30% reductions.

Kim et al. [7] present an iterative approach to the procedural abstraction techniques described by Debray et al. Procedural abstraction identifies sequences of instructions (instruction patterns) which are repeated and creates a procedure containing the pattern, replacing the original instances with a call to the procedure. In traditional procedural abstraction approaches, such as those discussed in the original paper [2], pattern identification is followed by a saving estimation before patterns are replaced. In contrast, the approach presented by Kim et al. iterates global saving estimation and pattern replacement following pattern identification. In eight benchmarks, an implementation targeting a CalmRISC8 embedded processor, averaged a 14.9% reduction in code size compared to traditional procedural abstraction techniques. However, due to the small number of instructions and registers on the CalmRISC8 processor, generated code would present many opportunities to the optimisation which could skew results higher than might be achieved on more common instruction sets.

Petrashko et al. [15] present context-sensitive call-graph construction algorithms which exploits the types provided to type parameters as contexts. They demonstrate through a motivational example that context-sensitive call graph analyses result in many calls being considered highly polymorphic and not able to be inlined. Through running analyses separately in the context of each possible type argument (possible through their context-sensitive call graph), the authors are able to remove all boxing and unboxing in addition to producing monomorphic calls which enable Java's JIT to inline and aggressively optimise functions. Petrashko et al. find that through use of the context-sensitive analysis, performance improves by 1.4x and discovers 20% more monomorphic call sites. Generated bytecode size was reduced by up to 5x and achieved the same performance on code as when hand-optimised with annotations.

2.0.3 Compilation Time Optimisations

Much like research which aims to reduce code-size, research into optimisations to reduce compilation time in more varied scenarios can be valuable to identify optimisation opportunities being missed.

Han et al. [5] discuss a technique for reducing compilation time in parallelising compilers. In this paper, the authors inspect the OSCAR compiler, which parallelises C or Fortran77 programs. OSCAR repeatedly applies multiple program analysis passes and restructuring passes. After OSCAR's second iteration, Han et al. identify that the OSCAR will apply an analysis to a function irrespective of whether the restructuring passes changed the function. Han et al. propose an optimisation to reduce compilation time by removing redundant analyses. Compilation time reductions of 27%, 13% and 19% are achieved for three large-scale proprietary real applications.

Leopoldseder et al. [9] present a technique to determine when it is beneficial to duplicate instructions from merge blocks to their predecessors in order to attain further optimisation. The authors implement their approach, Dominance-based duplication simulation (DBDS), in GraalVM JIT compiler, which introduces a constraint that the optimisation must consider the impacts on compilation time. Leopoldseder et al. identify that constant folding, conditional elimination, partial escape analysis/scalar replacement and read elimination are achievable after code duplication has taken place. Other code duplication approaches use backtracking to tentatively perform duplication while maintaining the option to reverse the duplication if it is determined not profitable. Backtracking approaches are compile-time intensive and thus not suitable for JIT compilation. Leopoldseder et al. choose to simulate duplication opportunities and then fit those opportunities into a cost model which maximises peak performance while minimising compilation time and code size. The authors achieve performance improvements of up to 40%, mean performance improvements of 5.89%, mean code size increases of 9.93% and mean compilation time increases of 18.84% across their benchmarks.

Unfortunately, none of these projects address the problem of reducing code-size when monomorphisation is used exclusively as a strategy for compiling generic code.

3. POLYMORPHISATION

Polymorphisation, a concept introduced by this paper, is an optimisation which determines when functions, closures and generators could remain polymorphic during code generation.

Rust supports parameterising functions by constants or types - these are known as generic functions and this feature is similar to generics in other languages, like Java's generics or C++'s templates. Generic functions and types are a desirable feature for programmers as they enable greater code reuse. When generating machine code, there are two approaches to dealing with generic functions - monomorphisation and boxing.

C++ and Rust perform monomorphisation, where multiple copies of a function are generated for each of the types or constants that the function was instantiated with. In contrast, Java performs dynamic dispatch, where each object is heap-allocated and a single copy of a function is generated, which takes an address.

In addition, Rust compiles to LLVM IR, the intermediate representation of LLVM. LLVM IR doesn't have any concept of generics, so Rust must perform either dynamic dispatch or monomorphisation.

The initial polymorphisation analysis implemented in this project determines when a type or constant parameter to a function, closure or generator is unused, and thus when this would result in multiple redundant copies of the function being generated. By generating fewer redundant monomorphisations of functions in the LLVM IR, there would be less work for LLVM to do, reducing compilation times and code size.

Types with unused generic parameters are disallowed by rustc, but there are no checks for unused generic parameters in functions. Despite this, it is assumed that it is rare for programmers to write functions which have unused generic parameters. However, closures inherit the generic parameters of their parent functions and often don't make use of these parameters.

For example, consider the code shown in Listing 1, which was taken from Serde, a popular Rust serialisation library.

```
1  fn parse_value<V>(
2      &mut self,
3      visitor: V,
4  ) -> Result<V::Value>
5  where
6      V: de::Visitor<'de>,
7  {
8      let peek = match self.parse_whitespace()? {
9          Some(b) => b,
10         None => return Err(
11             self.peek_error(
12                 ErrorCode::EofWhileParsingValue)),
13     };
14
15     let value = match peek {
16         // ...
17     };
18
19     match value {
20         Ok(value) => Ok(value),
21         Err(err) => Err(
22             err.fix_position(
23                 |code| self.error(code))),
24     }
25 }
```

Listing 1: Example from Serde

`parse_value` is a heavily used function which takes a single type parameter, `V`, and is instantiated many times with different types (a second type parameter from the surrounding block, `R`, is in scope too, which will also vary). This function contains one tiny closure on line 16 which would be monomorphised for each instantiation of `parse_value`. In Serde, instantiations of this closure contributed more LLVM IR than all but five larger functions in the library.

3.1 The Rust Compiler, rustc

This section introduces implementation details of rustc, the Rust compiler, which are necessary for the implementation of polymorphisation.

3.1.1 Query System

Traditionally, compilers are implemented in passes, where source code, an abstract syntax tree or intermediate representation is processed multiple times. Each pass takes the result of the previous pass and performs an analysis or operation - such as lexical analysis, parsing, semantic analysis, optimisation and code generation. Multi-pass compilers are well-established in the literature, but the expectations on compilers have changed in recent years.

Modern programming languages are expected to have high-quality integration into development environments [10], and assist in powering code completion engines and convenience features such as jump-to-definition. Moreover, users expect this information quickly and while they are writing their code (when it might not parse correctly or type-check). In addition, many programming languages' designs make it hard or impossible to statically determine the correct processing order, as expected by traditional multi-pass compiler architectures.

rustc's query system enables incremental and "demand-driven" compilation, and is integral to satisfying these expectations. Polymorphisation, as implemented in this project, uses the query system and limitations of the query system are reflected in its design (see Section 3.1.4).

In the query system, the compiler can be thought of as having a "database" of knowledge about the current crate, and queries are a mechanism for asking questions to the compiler. When compilation starts, the compiler's database is empty and is populated by queries when they are executed.

Queries consist of a name which identifies the query; a key that specifies what information is being requested; a result type; and a provider function that is executed when the result needs to be computed (i.e. it isn't already present in the database).

However, there are some restrictions:

- Query keys and results must be deeply immutable[3].
- Provider functions must be pure - the same input must always produce the same output.
- Provider functions must have two parameters - the query key and a reference to the query context (the rest of the database).

Query results are cached and if a query is invoked with the same key again, the result will be returned from the cache. Caching is critical for making the query system efficient and is why providers must be pure functions.

When no queries have been executed, the query system provides access to immutable input data. Since queries can only access other queries and the query context, without input data, queries would have no information to compute their result from.

At the start of compilation, rustc's driver creates a query context and invokes a top-level query, e.g. `compile`. `compile` would invoke other queries, such as `codegen-crate`, which would invoke further queries, eventually resulting in the

[3]Deeply immutable values do not have interior mutability. For example, if a query result were modified, then invoking the query again and retrieving the same query result from cache would not have those same modifications.

actual parsing from input data. As such, queries form a directed-acyclic graph.

Queries could form a cyclic graph by invoking themselves. To prevent this, the query engine checks for cyclic invocations and aborts execution.

As all queries are invoked through the query context, accesses can be recorded and a dependency graph can be produced. To avoid unnecessary computation when inputs change, the query utilises the dependency graph in the "red-green algorithm" used for query re-evaluation.

`try_mark_green` is the primary operation in the red-green algorithm. Before evaluating a query, the red-green algorithm checks each of the dependencies of the query. Each dependent query will have a colour: red, green or unknown. Green queries' inputs haven't changed, and the cached result can be re-used. Red queries' inputs have changed, and the result must be recomputed.

If the dependency's colour is unknown then `try_mark_green` is invoked recursively. Should the dependency be successfully marked as green, then the algorithm continues with the next of the current query's dependencies. However, if the dependency was marked as red, then the dependent query must be recomputed. When the result of the dependent query was the same as before re-execution then it would not invalidate the current node.

3.1.2 *Types and Substitutions*

rustc represents types with the `ty::Ty` type, which represents the semantics of a type (in contrast to `hir::Ty` from the HIR (high-level IR), rustc's desugared AST, which represents the syntax of a type).

`ty::Ty` is produced during type inference on the HIR, and then used for type-checking. `Ty` in the HIR assumes that any two types are distinct until proven otherwise, `u32` (Rust's 32-bit unsigned integer type) at line 10, column 20 would be considered different to `u32` at line 20, column 4.

`ty::Ty<'tcx>` is a type alias to `&'tcx TyS` (a type structure) where the primary implementation lives. Types are allocated in an arena in the global type context, where they are canonicalised and interned (allocating once for each distinct instance of the type). `TyS` has a `kind` field (of type `TyKind<'tcx>`) which is an enum defining all of the different kinds of types in the compiler. `Ty` is a recursive type which forms a tree and allows for representation of arbitrarily complex types.

Rust functions and types can be parameterised by generic parameters. Generic parameters can be types, constants or lifetimes. Generic parameters in Rust are similar to templates in C++ or generics in Java, and allow for a function or type to be used with many different concrete data types.

In rustc, polymorphic types are stored separately from the concrete types (or other generics) that are used with the type.
`SubstsRef<'tcx>` is used to represent the "substitutions" for a type (conceptually, a list of types that are to be substituted with the generic parameters of the type). `SubstsRef<'tcx>` is a type alias of `ty::List<GenericArg<'tcx>>`, where `GenericArg` is a space-efficient wrapper around `GenericArgKind`. `GenericArgKind` represents what kind of generic parameter is present - type, lifetime or const.

`Ty` contains `TyKind::Param` instances, each of which has a name and index. Only the index is required, the name is for debugging and error reporting. The index of the generic parameter is an integer which determines its order in the list of generic parameters in scope. To perform a substitution, the `subst` function on `Ty` is invoked with a `SubstsRef` which replaces each `TyKind::Param` with the type from the `SubstsRef` with the corresponding index.

The details of how generic parameters are implemented within rustc are directly leveraged by the polymorphisation implemented in this project, as well as the `TypeFoldable` infrastructure used to implement `susbts`.

`TypeFoldable` is a trait (Rust's equivalent of a Java interface) implemented by types that embed type information, allowing the recursive processing of the contents of the `TypeFoldable`. `TypeFolder` is another trait used in conjunction with `TypeFoldable`. `TypeFolder` is implemented on types (like `SubstsFolder`, which is used by `subst`) and is used to implement behaviour when a type, const, region or binder is encountered in a type. Used together, the `TypeFolder` and `TypeFoldable` traits allow for traversal and modification of complex recursive type structures.

3.1.3 *MIR*

MIR (mid-level IR) is an intermediate representation of functions, closures, generators and statics used in rustc, constructed from the HIR and type inference results. It is a simplified version of Rust that is better suited to flow-sensitive analyses. For example, all loops are simplified through use of a "goto" construct; match expressions are simplified by introducing a "discriminant" statement; method calls are replaced with explicit calls to trait functions; and drops/panics are made explicit.

MIR is based on a control-flow graph, composed of basic blocks. Each basic block contains zero or more statements with a single successor, and end with a terminator, which can have multiple successors.

Memory locations on the stack are known as "locals" in the MIR. Locals include function arguments, temporaries and local variables. Each local is identified by an index. The return value of a MIR body is stored into the local with index zero.

There are a handful of other key concepts in the MIR: "Places" are expressions which identify a location in memory; "Rvalues" are expressions which produce a value, these exist on the right-hand side of an assignment; "Operands" are arguments to rvalues, which can be constants or reads from places.

```rust
use std::collections::HashSet;

fn main() {
    let mut set = HashSet::new();
    set.insert(1);
    set.insert(2);
    set.insert(3);
}
```

Listing 2: Example for inspecting MIR

Within rustc, the MIR is a set of data structures, but can be dumped textually for debugging. Consider the MIR for the Rust code in Listing 2, it starts with the name and signature of the function in a pseudo-Rust syntax, shown in Listing 3.

Following the function signature, the variable declarations

```
// WARNING: This output format is intended for
// human consumers only and is subject to
// change without notice. Knock yourself out.
fn main() -> () {
    ...
}
```

Listing 3: Function prelude in MIR

for all locals are shown (Listing 4). Locals are printed with a leading underscore, followed by their index. Temporaries are intermingled with user-defined variables.

```
let mut _0: ();
let _2: bool;
let mut _3: &mut std::collections::HashSet<i32>;
let _4: bool;
let mut _5: &mut std::collections::HashSet<i32>;
let _6: bool;
let mut _7: &mut std::collections::HashSet<i32>;
```

Listing 4: Variables in MIR

To distinguish which locals are temporaries and which are user-defined variables, the MIR output then displays debuginfo for user-defined variables, such as `set`, shown in Listing 5.

Within each scope block, user-defined variables are listed with the user's name from the variable and the variable's place. In this example, the mapping is direct, but the compiler can store multiple user variables in a single local, and perform field accesses or dereferences.

Scope blocks represent the lexical structure of the source program (used in generating debuginfo). Each statement and terminator in the MIR output (omitted in these snippets) is followed by a comment which states which scope it exists in.

```
scope 1 {
    debug set => _1;
}
```

Listing 5: Scope in MIR

All of the remaining MIR output is used by the basic blocks of the current MIR body, starting with `bb0`, shown in Listing 6. Basic blocks are numbered from zero upwards, and contain statements followed by terminator on the last line.

`bb0` starts with a `StorageLive` statement, which indicates that the local `_1` is live (until a matching `StorageDead` statement, not included in this listing). Code generation uses `StorageLive` statements to allocate stack space. At the end of `bb0`, there is a `Call` terminator, which invokes `HashSet::new` and resumes execution in `bb2`. While terminators can have multiple successors, because `HashSet::new` does not require cleanup if it panicked, there is no panic edge.

`bb2` has two `StorageLive` statements followed by an `Assign` statement on line 4 into a temporary, creating a mutable borrow of `_1`, before ending the block with a `Call` terminator invoking `HashSet::insert`.

```
bb0: {
    StorageLive(_1);
    _1 = const
    ↪  std::collections::HashSet::<i32>::new() ->
    ↪  bb2;
}
```

Listing 6: bb0 in MIR

```
1  bb2: {
2      StorageLive(_3);
3      StorageLive(_3);
4      _3 = &mut _1;
5      _2 = const
       ↪  std::collections::HashSet::<i32>::insert(
       ↪  move _3, const 1i32) -> [return: bb3,
       ↪  unwind: bb4];
6  }
```

Listing 7: bb2 in MIR

There are other basic blocks in the MIR for Listing 2, but `bb0` and `bb2` are sufficient for understanding the key concepts of the MIR.

In addition to `StorageLive`, `Assign`, `StorageDead`, there are a selection of other statements:

- `Assign` statements write an rvalue into a place.
- `FakeRead` statements represent the reading that a pattern match might do, and exists to improve diagnostics.
- `SetDiscriminant` statements write the discriminant of an enum variant (its index) to the representation of an enum.
- `StorageLive` and `StorageDead` starts and ends the live range for storage of a local.
- `InlineAsm` executes inline assembly.
- `Retag` retags references in a place - this is part of the "Stacked Borrows" aliasing model by Jung et al. [6].
- `AscribeUserType` encodes a user's type ascription so that they can be respected by the borrow checker.
- `Nop` is a no-op, and is useful for deleting instructions without affecting statement indices.

Likewise, in addition to `Call`, there are various other terminators:

- `Goto` jumps to a single successor block.
- `SwitchInt` evaluates an operand to an integer and jumps to a target depending on the value.
- `Resume` and `Abort` indicate that a landing pad is finished and unwinding should continue or the process should be aborted.
- `Return` indicates a return from the function, assumes that `_0` has had an assignment.

- `Drop` and `DropAndReplace` drops a place (and optionally replaces it).
- `Call` invokes a function.
- `Assert` panics if a condition doesn't hold.
- `Yield` indicates a suspend point in a generator.
- `GeneratorDrop` indicates the end of dropping a generator.
- `FalseEdges` and `FalseUnwind` are used for control flow that is impossible, but required for borrow check to be conservative.

Closures - similar to anonymous functions or lambdas in other languages - which can capture variables from the parent environment, are also represented in the MIR. Functions and closures are represented almost identically in the MIR. Closures inherit the generic parameters of the parent function, and have additional generic parameters tracking the closure's inferred kind, signature and upvars (captured variables).

Generators are suspendable functions, which operate similarly to closures, but can yield values in addition to returning them. Like closures, generators are represented almost identically to functions in the MIR, but have extra generic parameters which track the generator's inferred resume type, yield type, return type and witness (an inferred type which is a tuple of all types that could up in a generator frame).

rustc provides a pair of visitor traits (depending on whether mutability is required) which types can implement to simplify traversal of the MIR.

3.1.4 Shims

There are some circumstances where rustc will generate functions during compilation, these functions are known as shims, and they are often specialised for specific types.

Drop glue is an example of a shim generated by the compiler. Rust doesn't require that the user write a destructor for their type, any type will be dropped recursively in a specified order. To implement this, rustc transforms all drops into a call to `drop_in_place<T>` for the given type.

`drop_in_place`'s implementation is generated by the compiler for each `T`. There are a variety of shims generated by the compiler:

- Drop shims implements the drop glue required to deallocate a given type.
- Clone shims implement cloning for a builtin type, like arrays, tuples and closures.
- Closure-once shims generate a call to an associated method of the `FnMut` trait.
- Reify shims are for `fn()` pointers where the function cannot be turned into a pointer.
- `FnPtr` shims generate a call after a dereference for a function pointer.
- `Vtable` shims generate a call after a dereference and move for a vtable.

During code generation, the MIR being translated into LLVM IR (or Cranelift IR) has no substitutions applied, therefore all of the generic parameters in the function are still present. Where required, substitutions (those collected during monomorphisation, as will be discussed in Section 3.1.5) are applied to types. However, for shims, these substitutions are typically redundant as the MIR of the shim is generated specifically for a type, and do not have any remaining generic parameters.

3.1.5 Monomorphisation

In order to determine which items will be included in the generated LLVM IR for a crate, rustc performs monomorphisation collection. Non-generic, non-const functions map to one LLVM artefact, whereas generic functions can contribute zero to `N` artefacts depending on the number of instantiations of the function. Monomorphisations can be produced from instantiations of functions defined in other crates.

"Mono items" are anything that results in a function or global in the LLVM IR of a codegen unit. Mono items can reference other mono items, for example, if a function `baz` references a function `quux` then the mono item for `baz` will reference the mono item for `quux` (typically this results in the generated LLVM IR for `baz` referencing the generated LLVM IR for `quux` too). Therefore, mono items form a directed graph, known as the "mono item graph", which contains all items necessary for codegen of the entire program.

In order to compute the mono item graph, the collector starts with the roots - non-generic syntactic items in the source code. Roots are found by walking the HIR of the crate, and whenever a non-generic function, method or static item is found, a mono item is created with the `DefId` of the item and an empty `SubstsRef` (since the item is non-generic).

From the roots, neighbours are discovered by inspecting the MIR of each item. Whenever a reference to another mono item is found, it is added to the set of all mono items. This process is repeated until the entire mono item graph has been discovered. By starting with non-generic roots, the current mono item will always be monomorphic, so all generic parameters of neighbours will always be known. References to other mono items can take the form of function calls, taking references to functions, closures, drop glue, unsizing casts and boxes.

```rust
fn main() {
    bar(2u64);
}

fn foo() {
    bar(2u32);
}

fn bar<T>(t: T) {
    baz(t, 8u16);
}

fn baz<F, G>(f: F, g: G) {
}
```

Listing 8: Monomorphisation Example

For example, consider the code in Listing 8. Monomorphisation will start by walking the HIR of the crate and looking for non-generic syntactic items. After this step, the set of mono items will contain the functions `main` and `foo`, both of which have no generic parameters.

Next, monomorphisation will traverse the MIR of each root, looking for neighbours. In `main`, the terminator of `bb0` (shown in Listing 9) references the `bar` function, with the substitutions `u64`.

```
bb0: {
    StorageLive(_1);
    _1 = const bar::<u64>(const 2u64) -> bb1;
}
```

Listing 9: bb0 of main

This will result in a mono item for `bar` being added to the set of mono items, and being visited for neighbours. `foo` was also determined to be a root, so its MIR (shown in Listing 10) will also be visited to find neighbours, resulting in another mono item for `bar` being added to the set, with the substitution `u32`.

```
bb0: {
    StorageLive(_1);
    _1 = const bar::<u32>(const 2u32) -> bb1;
}
```

Listing 10: bb0 of foo

`bar`'s MIR (shown in Listing 11) will be visited twice, once for `u32` and once for `u64`, each time adding `baz` to the set of mono items with different substitutions.

```
bb0: {
    StorageLive(_2);
    StorageLive(_3);
    _3 = move _1;
    _2 = const baz::<T, u16>(move _3, const 8u16)
    ↪  -> bb1;
}
```

Listing 11: bb0 of bar

Finally, `baz` will be visited twice for neighbours, with substitutions `u32, u16` and `u64, u16`, but as it does not reference any other mono items, it does not add any new mono items to the set.

3.2 Analysis

Implementation of this project first requires a functioning polymorphisation analysis to determine when generic parameters are unused. To do so, a new query is introduced, `used_generic_params` (as shown in Listing 12), which takes a `DefId` identifying the item being analysed - e.g. the function, method, closure or generator - and returns a `BitSet`.

`used_generic_params`'s return type has a domain size equal to the number of generic parameters in scope for the item being analysed. The analysis starts by returning early for items which have no generic parameters in scope or where there is no MIR available (e.g. trait methods without a default implementation), this is shown in Listing 13.

```
query used_generic_params(key: DefId) ->
↪  BitSet<u32> {
    desc {
        |tcx| "determining which generic
↪  parameters are used by `{}`",
↪  tcx.def_path_str(key)
    }
}
```

Listing 12: Query definition

```
fn used_generic_params(
    tcx: TyCtxt<'_>,
    def_id: DefId,
) -> BitSet<u32> {
    let generics = tcx.generics_of(def_id);
    let count = generics.count();
    // Exit early when there are no parameters to
    // be unused.
    if count == 0 {
        return BitSet::new_filled(count);
    }
    // Exit early when there is no MIR available.
    if !tcx.is_mir_available(def_id) {
        return BitSet::new_filled(count);
    }
    // ...
}
```

Listing 13: Trivial cases

As described in Section 3.1.2, closures and generators have additional "synthetic" generic parameters. These parameters must be considered used for code-generation to succeed and do not negatively impact the effectiveness of the project. In addition, lifetimes are represented as generic parameters like type and constant parameters, and these should always be marked as used as they do not impact monomorphisation.

This is performed by a `mark_used_by_default_parameters` function (shown in Listing 14) which takes the `DefId` of an item and its generics.

`ty::Generics` is the result of calling the `generics_of` query and has a `params` field containing vector of generic parameters for the current item. The generics of the parent, despite being in scope, are not included in this vector. `ty::Generics` also has a `parent` field with the `DefId` of the parent item (if there is one), `generics_of` can be invoked on that to get the parent parameters.

For example, while a closure inherits the generic parameters of its parent item (they are in scope), `generics.params` only contains the generic parameters defined on the closure (only the synthetic parameters), and `generics.parent` contains the `DefId` of the parent item with the real generic parameters.

This structure is taken advantage of by `mark_used_by_default_parameters`. By checking if a item is a closure or generator, it can mark all parameters as used indiscriminately, falling back to only-lifetime-parameters otherwise. `mark_used_by_default_parameters` proceeds to invoke itself recursively with the parent's generics when they exist.

Next, the analysis calls the `optimized_mir` query to get

```rust
fn mark_used_by_default_parameters<'tcx>(
    tcx: TyCtxt<'tcx>,
    def_id: DefId,
    generics: &'tcx ty::Generics,
    used_parameters: &mut BitSet<u32>,
) {
    if !tcx.is_trait(def_id)
       && (tcx.is_closure(def_id)
           || tcx.type_of(def_id).is_generator())
    {
        for param in &generics.params {
            used_parameters.insert(param.index);
        }
    } else {
        for param in &generics.params {
            if param.is_lifetime() {
                used_parameters.insert(
                    param.index);
            }
        }
    }

    if let Some(parent) = generics.parent {
        mark_used_by_default_parameters(
            tcx, parent, tcx.generics_of(parent),
            used_parameters);
    }
}
```

Listing 14: Used-by-default parameters

the MIR body for the current item and uses a MIR visitor to traverse it. Only three components of the MIR need to be visited by the analysis: locals, types and constants.

`visit_local_decl` is invoked for each local variable in the item's MIR before the MIR is traversed. This analysis' implementation of `visit_local_decl` calls `super_local_decl` (which proceeds with traversal as normal) for all but one case: If the current item is a closure or generator and if the local currently being visited is the first local, then it returns early.

In the MIR, the first local for a closure or generator is a reference to the closure of generator itself. If the analysis proceeded to visit this local as normal, then it would eventually visit the substitutions of the closure or generator. Unfortunately, this would result in all inherited generic parameters for a closure *always* being considered used, defeating the point of the analysis.

`visit_const` and `visit_ty` are invoked for each constant and type in the MIR, and are where the visitor "bottoms out". This analysis' implementation of `visit_const` and `visit_ty` call a `visit_foldable` function (which isn't part of the MIR visitor).

`visit_foldable` takes any type which implements `TypeFoldable` and checks if the type has any type or constant parameters before visiting it with this analysis' type visitor.

In the type visitor, the analysis traverses the structure of a type or constant and looks for `ty::Param` and `ConstKind::Param`, setting the index from each within the `BitSet`, marking the parameter as used. In addition, the type visitor also special-cases closures and generators. Instead of traversing the closure or generator type, the analysis is invoked recursively on the closure or generator, and any parameters from the parent which were used in the closure or generator are marked as used in the parent.

The analysis was significantly more complex during implementation and had to special-case some casts, call and drop terminators to avoid marking parameters as used unnecessarily, often in order to avoid parts of the compiler which were previously always monomorphic. However, during integration of the analysis, as support for polymorphic MIR during codegen was improved, and these special-cases were removed.

Some generic parameters are only used in the bounds of another generic parameter. After the analysis has detected which generic parameters are used in the body of the item, the predicates of the item are checked for uses of unused generic parameters in the bounds on used generic parameters.

```rust
fn bar<I>() {}

fn foo<I, T>(_: I)
where
    I: Iterator<Item = T>,
{
    bar::<I>()
}
```

Listing 15: Example of generic parameter use in predicate

Consider the example shown in Listing 15. `foo` has two generic parameters, `I` and `T`, and uses `I` in the substitutions of a call to `bar`. If only the body of `foo` was analysed then `T` would be considered unused despite it being clear that `T` is used in the iterator bound on `I`.

3.2.1 Testing

To test of the analysis, a debugging flag, `-Z polymorphize-errors`, is added, which emits compilation errors on items with unused generic parameters, an example of which is shown in Listings 16 and 17.

By emitting compilation "errors", the project leverages the existing error diagnostics infrastructure to enable introspection into the compiler internals from a test written at the Rust source-code level.

```rust
struct Foo<F>(F);

impl<F: Default> Foo<F> {
    // Function has an unused generic
    // parameter from impl.
    pub fn unused_impl() {
//~^ ERROR item has unused generic parameters
    }
}
```

Listing 16: Example of an analysis test

In addition, this approach integrates well into rustc's existing test infrastructure, which writes all of the error output for a test into a `stderr` file and checks that the output hasn't changed by comparison with that file. In addition, anywhere an error is expected is annotated in the test source.

```
error: item has unused generic parameters
  --> $DIR/functions.rs:38:12
   |
LL | impl<F: Default> Foo<F> {
   |      - generic parameter `F` is unused
LL |     // Function has an unused generic
LL |     // parameter from impl.
LL |     pub fn unused_impl() {
   |            ^^^^^^^^^^^
```

Listing 17: Example of an analysis test error

3.3 Implementing polymorphisation in rustc

Integration of the analysis' results into the compiler required significant debugging and many targeted modifications where previously only monomorphic types and MIR were expected.

Throughout rustc's code generation infrastructure, the `ty::Instance` type is used. `Instance` is constructed through use of `Instance::resolve` which takes a `(DefId, SubstsRef<'tcx>)` pair and returns an `Instance` only if there is enough information to identify a specific function which will be generated. In a monomorphic context, after coherence and type-checking, `Instance::resolve` will always succeed.

`Instance::polymorphize` is introduced to apply the results of the `used_generic_params` analysis to an `Instance`. For each parameter in a `SubstsRef<'tcx>`, the matching index in the analysis' `BitSet` is checked to determine whether the parameter was used. If the parameter was used, then the current parameter is kept. If the parameter was unused, then the parameter is replaced with an identity substitution. During implementation, replacing the parameter with a new `ty::Unused` type was considered, but it was hypothesised that this would require more changes to the compiler than an identity substitution.

For example, given an unsubstituted type containing `ty::Param(0)`, if the parameter were considered used by the analysis, then it would be substituted by the real type, but, if the parameter were considered unused, then it would be substituted with `ty::Param(0)` and remain the same.

During monomorphisation collection, as described in Section 3.1.5, when a mono item is identified, it is added to a set. This project concerns itself with one kind of mono item, a `MonoItem::Fn` which contains an `Instance` type. When a new `MonoItem::Fn` is created, `Instance::polymorphize` is invoked before it is added to the final mono item set. This has the effect of reducing the number of mono items.

```
fn foo<A, B>(_: B) { }

fn main() {
    foo::<u64, u32>(1);
    foo::<u32, u32>(2);
    foo::<u16, u32>(3);
}
```

Listing 18: Example of an polymorphisation deduplication

Consider the example in Listing 18, monomorphisation would normally produce three mono items (and three copies of the function):

- `foo::<u16, u32>`
- `foo::<u32, u32>`
- `foo::<u64, u32>`

However, with the analysis, only a single mono item would be produced: `foo::<T, u32>`.

rustc's code generation iterates over the mono items collected and generates LLVM IR for each, adding the result to an LLVM module. By de-duplicating during monomorphisation collection, polymorphisation "falls out" of the way the compiler is structured (though some more changes are required).

In particular, various changes to code generation were required wherever another `Instance` is referenced. For example, when generating LLVM IR for a `TerminatorKind::Call` (a `call` or `invoke` instruction in LLVM), an `Instance` is generated to determine the exact item which is being invoked and its mangled symbol name. Without applying the `Instance::polymorphize` function, the instruction would refer to the mangled symbol for the unpolymorphised function, which no longer exists.

3.3.1 *Upstream monomorphisations*

rustc will avoid performing monomorphisation of an item from an upstream crate if the compiled upstream crate already contains a monomorphisation for the given substitutions and it can just be linked to.

Whether monomorphisation should be performed locally is handled by a `should_codegen_locally` function which checked whether upstream monomorphisations where available for a given `Instance`. `should_codegen_locally` is invoked before a mono item is created (and polymorphisation occurs), so it looks for an unpolymorphised monomorphisation in the upstream crate. However, if the item would be polymorphised then the upstream crate would only contain a polymorphised copy, which would result in unnecessary code generation when compiling the current crate. `should_codegen_locally` was changed to always check for a polymorphised monomorphisation in the upstream crate.

3.3.2 *Specialisation*

In Rust, data structures are defined separately from their methods. Multiple `impl` blocks can exist which implement methods on a type. For generic types, `impl` blocks can pick concrete types for parameters or remain generic, but multiple `impls` cannot overlap. Specialisation is an unstable feature in Rust which enables `impl` blocks to overlap if one block is clearly "more specific" than another.

`Instance::resolve` can only resolve specialised functions of traits under certain circumstances and detects whether further specialisation can occur by whether the item needs substitution (i.e. has unsubstituted type or constant parameters). After integration of polymorphisation, this check had to be more robust. In particular, polymorphisation meant that closure types in `Instance::resolve` could need substitution (by containing a `ty::Param` from an unused parent parameter) and would thus "need substitution" for the purposes of the specialisation check, despite this having no impact on whether the item was further specialisable.

Initially, this was resolved by implementing a fast type visitor which checked for generic parameters, but would not visit the parent substitutions of a closure or generator. However, this check is in the "fast path" and a visitor would have been too expensive, so this was replaced by a addition to the `TypeFlags` infrastructure within the compiler (after some refactoring [1] enabled this change), shown partially in Listing 19.

```rust
// ...
&ty::Closure(_, ref substs) => {
    let substs = substs.as_closure();
    let should_remove_further_specializable =
        !self.flags.contains(
            STILL_FURTHER_SPECIALIZABLE);
    self.add_substs(substs.parent_substs());
    if should_remove_further_specializable {
        self.flags -= STILL_FURTHER_SPECIALIZABLE;
    }

    self.add_ty(substs.sig_as_fn_ptr_ty());
    self.add_ty(substs.kind_ty());
    self.add_ty(substs.tupled_upvars_ty());
}
// ...
```

Listing 19: Snippet of type flags infrastructure

3.3.3 *Shims*

As described in Section 3.1.4, rustc generates shim functions to support language features like dropping.

rustc's infrastructure surrounding shims and code generation required that `Instance`s were monomorphic. This would cause problems for the interaction between polymorphisation and all sorts of shims, but drop glue is the simplest (Listing 20 has a minimised test case which would reproduce this issue with earlier versions of the implementation).

```rust
pub struct OnDrop<F: Fn()>(pub F);

impl<F: Fn()> Drop for OnDrop<F> {
    fn drop(&mut self) { }
}

fn foo<R, S: FnOnce()>(
    _: R,
    _: S,
) {
    let bar = || {
        let _ = OnDrop(|| ());
    };
    let _ = bar();
}

fn main() {
    foo(3u32, || {});
}
```

Listing 20: Test case for polymorphisation and drop shims

`Instance` types are composed of two parts - a `SubstsRef<'tcx>` and a `InstanceDef`. For normal functions, `InstanceDef::Item(DefId)` is the variant of `InstanceDef` which is used. However, for drop glue, a `InstanceDef::DropGlue(DefId, Option<Ty<'tcx>>)` variant is used, where the `Ty<'tcx>` is the type that the shim is being generated to drop. `InstanceDef::DropGlue`'s `DefId` always referred to the `drop_in_place` function.

With polymorphisation, the `Ty<'tcx>` can contain unsubstituted parameters and that type ends up in the generated MIR for the shim. Then, during code generation, the `SubstsRef<'tcx>` which contains the same identity parameters would be applied to the `Ty<'tcx>`, resulting in double substitution (which causes an "substitution failure" internal compiler error).

For example, `drop_in_place::<Vec<T>>`'s shim produces a MIR body referring to `Vec<T>` but which also is going to get monomorphised by `{ T -> Vec<T> }`, resulting in `Vec<Vec<T>>` which is nonsense.

Listing 20 highlights this issue where a closure (which contains reference to unused generic parameters from the parent) is moved into a type which implements `Drop` and would result in a drop shim being generated.

This case could be detected in the analysis by special-casing `TerminatorKind::Drop`, but this breaks down when the test case becomes more complex, as shown in Listing 21. In Listing 21, the closure is defined and analysed in `foo` (correctly determining that no parent parameters are used), but referenced by a type which implements `Drop` another function. To detect this case, the analysis would need to be made transitive, but this is not possible due to limitations of the query system regarding cycles (as described in Section 3.1.1).

```rust
pub struct OnDrop<F: Fn()>(pub F);

impl<F: Fn()> Drop for OnDrop<F> {
    fn drop(&mut self) { }
}

fn bar<F: FnOnce()>(f: F) {
    let _ = OnDrop(|| ());
    f()
}

fn foo<R, S: FnOnce()>(
    _: R,
    _: S,
) {
    let bar = || {
        bar(|| {})
    };
    let _ = bar();
}

fn main() {
    foo(3u32, || {});
}
```

Listing 21: Transitive test case for polymorphisation and drop shims

Preliminary work to remove the requirement that `Instances` be monomorphic was completed in PR #69935 [25] which resolved this issue without modification of the analysis.

4. EVALUATION

To evaluate the project, the performance of the compiler on a set of benchmarks is compared with a build of the compiler without this project's changes applied.

For the purposes of evaluation, the compiler is built in release configuration from the Git commit with the hash `2f2ccd2` [26] (PR #69749 [28] at `9ef1d94` [27] merged with master, `150322f` [8]). `150322f` is also built in release configuration to act as a baseline for comparisons.

Benchmarking has been performed two times previously and performance improvements identified from those earlier runs are implemented in the versions being compared in this section, as discussed in Section 4.1.

There are 40 benchmarks that are run in rustc's standard performance benchmarking suite [29]. Benchmarks are either real Rust programs from the ecosystem which are important; real Rust programs from the ecosystem which are known to stress the compiler; or artificial stress tests.

All benchmarks are compiled with both debug and release profiles, some benchmarks are run with the check profile. Each benchmark is run at least four times:

- **clean**: a non-incremental build.
- **baseline incremental**: an incremental build starting with empty cache.
- **clean incremental**: an incremental build starting with complete cache, and clean source directory – the "perfect" scenario for incremental.
- **patched incremental**: an incremental build starting with complete cache, and an altered source directory. The typical variant of this is "println" which represents the addition of a `println!` macro somewhere in the source code.

`perf` is used to collect information on the execution of each benchmark, notably: CPU clock; clock cycles; page faults; memory usage; instructions executed; task clock and wall time. In addition, for each run, the execution count and total time spent on each query in rustc's query system is also captured - this allows for detailed comparisons between runs to better determine why there has been a performance impact.

Furthermore, for a subset of the benchmarks which had an appreciable performance impact, the number of mono-items generated are compared.

Benchmarks were performed on a Linux host with an AMD Ryzen 5 3600 6-Core Processor and 64GB of RAM with ASLR disabled in both the kernel and userspace.

4.1 Discussion

For a majority of the benchmarks, there was no appreciable difference in compilation time performance. This result is expected, the optimisation would have limited impact in programs which do not contain highly generic code containing closures.

In those benchmarks that did not show any significant improvement or regression, any performance differences between the compiler versions are considered noise. The threshold used for noise is determined by comparing compiler versions that had no functional changes ("noise runs") and observing the impact on performance results on each benchmark.

Three benchmarks were appreciably impacted by the optimisation implemented in this PR, results for these in Table 1 (complete results available online [29]). The number of mono items generated for these benchmarks was also compared and the results are shown in Table 2.

- `script-servo` is the `script` crate from Servo, the parallel browser engine used by Firefox.
- `regression-31157` is a small program which caused large performance regressions in earlier versions of the Rust compiler.
- `ctfe-stress-4` is a small program which is used to stress compile-time function evaluation.

In applicable benchmarks, compilation time performance on clean runs generally improves by 3-5%. Incremental runs generally have lower performance improvements, if any. `script-servo` in the release configuration's results show a 11.4% slowdown in patched incremental runs. Query profiling shows that this slowdown comes from LLVM LTO's optimisations - this could be due to LTO being more effective and thus more optimisation taking place or that the polymorphised functions make LTO more expensive to perform. However, LTO does not appear to impact performance in the debug configuration or on other benchmarks. `script-servo` in the debug configuration shows a 5.1% performance improvement which corresponds to an 11 second decrease in compilation time.

`ctfe-stress-4`'s mono-item count shows no reduction. This isn't surprising as the test contains mostly compile-time evaluated functions (as expected for a test which aims to stress compile-time function evaluation). This suggests that that polymorphisation may have reduced the number of functions which had to be evaluated at compile-time.

Memory usage across the benchmarks was very varied, ranging from 19% increases to 15% decreases, depending on the benchmark. Like instructions executed, memory usage increases typically happened when incremental compilation was enabled, while decreases happened when incremental compilation was disabled.

Earlier benchmarking revealed that re-execution of the `used_generic_params` query resulted in increased loads from crate metadata (intermediate data from compilation of dependencies) so the version of the query benchmarked in this paper's results were added to crate metadata and incremental compilation storage. In addition, the `BitSet` returned from the query was replaced with a `u64` as a micro-optimisation.

5. CONCLUSIONS AND FUTURE WORK

This paper has presented a compilation time optimisation in the Rust compiler which reduces redundant monomorphisation. Currently, this implementation from this project is being reviewed as PR #69749 [28] in the upstream Rust project and preliminary work, such as PR #69935 [25], has landed.

There are no aspects of the Rust language itself which makes implementation of this optimisation particularly challenging or complex, most of the difficulty arises from implementation details of rustc. With that said, rustc is well suited to having this optimisation implemented.

Due to the structure of the compiler, in an ideal scenario, only modifications to monomorphisation and call generation are necessary to enable polymorphisation. Alternate approaches, such as changing the the MIR of closures and generators to only inherit the parameters that they use from their parents, would be significantly more invasive and challenging, as the current system has emergent properties which enables simplifications in other areas of the compiler.

However, rustc will not always fail eagerly when an invalid case is encountered, which can significantly lengthen debugging sessions as the root cause of a failure is determined. This project has provided an excellent opportunity to learn more about the structure of the Rust compiler, and the techniques used in modern, production compilers to implement LLVM IR generation.

Implementation of polymorphisation resulted in 3-5% compilation time improvements in some benchmarks. While there are further improvements that could be made to improve the integration and improve performance when incremental compilation is enabled (see Section 5.1), this is a promising result.

5.1 Future Work

Investigation into the interaction between polymorphisation and LTO should be performed to determine the source of the compile-time regression noticed in the `script-servo` benchmark. In addition, tweaking of the circumstances under which the query's results are cached (both incrementally and cross-crate) should be performed to impact on performance experienced when incremental compilation is enabled.

With infrastructure in place to support polymorphic code generation, the polymorphisation analysis can be extended to detect more advanced cases. For example, when only the size of a type is used then monomorphisation can occur for the distinct sizes of instantiated type; or when the memory representation of a type is independent of the representation of its generic parameters, such as `Vec<T>` (because the `T` is behind indirection), functions like `Vec::len` can be polymorphised.

Furthermore, the integration currently doesn't fully work with Rust's new symbol mangling scheme, which isn't yet enabled by default - support for this could be added.

6. ACKNOWLEDGEMENTS

Thanks to Eduard-Mihai Burtescu, Niko Matsakis and everyone else in the Rust compiler team for their assistance and time during this project. In addition, thanks to Michel Steuwer for his comments and guidance throughout the year.

7. REFERENCES

[1] E.-M. Burtescu. rustc: keep upvars tupled in Closure,Generatorsubsts., 2020 (accessed March 25, 2020). `https://github.com/rust-lang/rust/pull/69968`.

[2] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, Mar. 2000.

[3] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, page 42–47, New York, NY, USA, 2009. Association for Computing Machinery.

[4] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta. Exploiting function similarity for code size reduction. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, page 85–94, New York, NY, USA, 2014. Association for Computing Machinery.

[5] J. Han, R. Fujino, R. Tamura, M. Shimaoka, H. Mikami, M. Takamura, S. Kamiya, K. Suzuki, T. Miyajima, K. Kimura, and et al. Reducing parallelizing compilation time by removing redundant analysis. In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*, SEPS 2016, page 1–9, New York, NY, USA, 2016. Association for Computing Machinery.

[6] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer. Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

[7] D.-H. Kim and H. J. Lee. Iterative procedural abstraction for code size reduction. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, page 277–279, New York, NY, USA, 2002. Association for Computing Machinery.

[8] T. R. P. Language. commit 150322f, 2020 (accessed March 25, 2020). `https://github.com/rust-lang/rust/commit/150322f86d441752874a8bed603d71119f190b8b`.

[9] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. Dominance-based duplication simulation (dbds): Code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 126–137, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Microsoft. Language server protocol, 2020 (accessed March 29, 2020). `https://microsoft.github.io/language-server-protocol/`.

[11] S. Overflow. Stack overflow developer survey 2016, 2016 (accessed March 25, 2020). `https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted`.

[12] S. Overflow. Stack overflow developer survey 2017, 2017 (accessed March 25, 2020). `https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted`.

[13] S. Overflow. Stack overflow developer survey 2018, 2018 (accessed March 25, 2020). `https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted`.

[14] S. Overflow. Stack overflow developer survey 2019, 2019 (accessed March 25, 2020). `https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-languages`.

[15] D. Petrashko, V. Ureche, O. Lhoták, and M. Odersky. Call graphs for languages with parametric polymorphism. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 394–409, New York, NY, USA, 2016. Association for Computing

Machinery.

[16] J. Stone. Reduce the genericity of closures in the iterator traits, 2019 (accessed December 3, 2019). https://github.com/rust-lang/rust/pull/62429.

[17] J. Stone. Reduce the genericity of many closures, 2019 (accessed December 3, 2019). https://github.com/rayon-rs/rayon/pull/673.

[18] N. Stucki and V. Ureche. Bridging islands of specialized code using macros and reified types. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery.

[19] T. R. S. Team. State of rust survey 2018, 2018 (accessed March 25, 2020). https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html.

[20] T. R. S. Team. State of rust survey 2019 (wip), 2020 (accessed March 25, 2020). https://github.com/rust-lang/blog.rust-lang.org/pull/544.

[21] D. Tolnay. Instantiate fewer copies of a closure inside a generic function, 2017 (accessed December 3, 2019). https://github.com/rust-lang/rust/issues/46477.

[22] J. Turner. State of rust survey 2017, 2017 (accessed March 25, 2020). https://blog.rust-lang.org/2017/09/05/Rust-2017-Survey-Results.html.

[23] V. Ureche, M. Stojanovic, R. Beguet, N. Stucki, and M. Odersky. Improving the interoperation between generics translations. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, page 113–124, New York, NY, USA, 2015. Association for Computing Machinery.

[24] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the speed to code size tradeoff in parametric polymorphism translations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 73–92, New York, NY, USA, 2013. Association for Computing Machinery.

[25] D. Wood. codegen/mir: support polymorphic ‘instancedef‘s, 2020 (accessed March 25, 2020). https://github.com/rust-lang/rust/pull/69935.

[26] D. Wood. commit 2f2ccd2, 2020 (accessed March 25, 2020). https://github.com/rust-lang/rust/commit/2f2ccd26f9e4af93df8495c3d7a2fec418590f97.

[27] D. Wood. commit 9ef1d94, 2020 (accessed March 25, 2020). https://github.com/rust-lang/rust/commit/9ef1d943be1850e1f486ab9732e545dfd99d1fc8.

[28] D. Wood. Polymorphization, 2020 (accessed March 25, 2020). https://github.com/rust-lang/rust/pull/69749.

[29] D. Wood. rustc performance data, 2020 (accessed March 29, 2020). https://perf.rust-lang.org/compare.html?start=150322f86d441752874a8bed603d71119f190b8b&end=2f2ccd26f9e4af93df8495c3d7a2fec418590f97.

Table 1: Compile-time Performance Results (Instructions Executed)

Benchmark	**Profile**	**Average**	**Minimum**	**Maximum**
	Mode	**Instructions** (`150322f`)	**Instructions** (`2f2ccd2`)	**Difference** (%)
`script-servo`	Release	1.4%	-2.4%	11.4%
	Clean	2,055,023,148,841	2,005,849,839,832	-2.4%
	Baseline Incremental	3,121,194,565,069	3,077,410,967,110	-1.4%
	Clean Incremental	90,681,473,264	90,853,410,015	0.2%
	Patched Incremental (`println!` in dependency)	1,095,860,923,599	1,117,792,391,471	2.0%
	Patched Incremental (`println!`)	442,478,442,625	492,780,600,788	11.4%
	Patched Incremental (commit `8b0f58c8a`)	3,043,338,800,878	2,998,036,491,796	-1.5%
`script-servo`	Debug	-1.4%	-5.1%	1.8%
	Clean	667,901,459,840	633,560,959,928	-5.1%
	Baseline Incremental	1,523,791,662,564	1,487,014,053,598	-2.4%
	Clean Incremental	95,788,373,594	95,656,537,038	-0.1%
	Patched Incremental (`println!` in dependency)	109,780,442,755	111,773,769,666	1.8%
	Patched Incremental (`println!`)	101,164,767,648	100,959,856,263	-0.2%
	Patched Incremental (commit 8b0f58c8a)	1,340,727,740,661	1,307,554,829,275	-2.5%
`regression-31157`	Release	1.8%	0.7%	3.4%
	Clean	20,909,557,684	21,060,032,968	0.7%
	Baseline Incremental	19,035,673,820	19,253,666,422	1.1%
	Clean Incremental	635,341,137	647,334,533	1.9%
	Patched Incremental (`println!`)	7,799,412,422	8,065,114,525	3.4%
`ctfe-stress-4`	Check	-1.8%	-2.9%	0.1%
	Clean	53,140,621,042	51,592,420,003	-2.9%
	Baseline Incremental	71,333,125,770	69,431,948,215	-2.7%
	Clean Incremental	4,862,403,451	4,865,044,498	0.1%
`ctfe-stress-4`	Debug	-1.8%	-2.8%	0.1%
	Clean	54,487,817,509	53,050,150,847	-2.8%
	Baseline Incremental	72,770,459,110	70,890,609,287	-2.6%
	Clean Incremental	4,876,267,173	4,879,015,542	0.1%
`ctfe-stress-4`	Release	-1.8%	-2.8%	0.1%
	Clean	54,675,367,364	53,147,068,455	-2.8%
	Baseline Incremental	72,722,515,906	70,858,424,696	-2.6%
	Clean Incremental	4,876,551,941	4,879,084,682	0.1%

Table 2: Mono Item Results

benchmark	**profile**	**count** (`150322f`)	**count** (`2f2ccd2`)
`script-servo`	release	213,642	207,642
`script-servo`	debug	194,509	188,763
`regression-31157`	release	3635	3608
`regression-31157`	debug	2905	2881
`ctfe-stress-4`	release	13	13
`ctfe-stress-4`	debug	13	13