Polymorphisation: Improving Rust's compilation times through intelligent monomorphisation

David Wood (2198230W)

MSci Software Engineering with Work Placement (40 Credits)

What is Rust?

Rust is a new systems programming language, designed to provide memory safety while maintaining high performance.

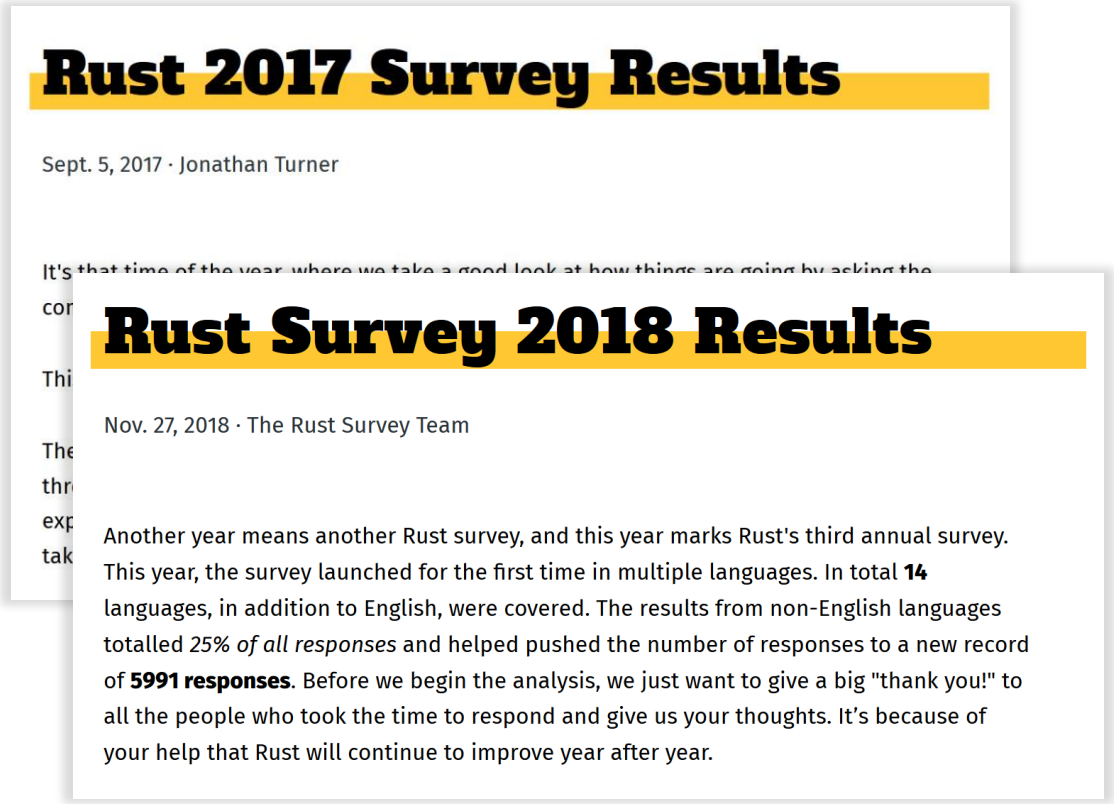
Rust has been voted “most loved programming language” in Stack Overflow’s Developer Survey for four years.

It is used at many well known companies, including Atlassian, Braintree, Canonical, Chef, Google and Deliveroo.

Rust's Compilation Speed

In the yearly "State of Rust" survey, users of the language have requested improved compilation speed since 2017.

There are a handful of reasons why Rust is slow to compile, and this project aims to tackle one of them – monomorphisation.



Rust 2017 Survey Results

Sept. 5, 2017 · Jonathan Turner

It's that time of the year, where we take a good look at how things are going by asking the cor

Thi

The
thr
exp
tak

Rust Survey 2018 Results

Nov. 27, 2018 · The Rust Survey Team

Another year means another Rust survey, and this year marks Rust's third annual survey. This year, the survey launched for the first time in multiple languages. In total **14** languages, in addition to English, were covered. The results from non-English languages totalled *25% of all responses* and helped pushed the number of responses to a new record of **5991 responses**. Before we begin the analysis, we just want to give a big "thank you!" to all the people who took the time to respond and give us your thoughts. It's because of your help that Rust will continue to improve year after year.

Monomorphisation

Monomorphisation is when generic code is duplicated for every instantiation.

Runtime performance is often improved by monomorphisation, but it results in more work being necessary at compile-time.

Monomorphisation is necessary as LLVM IR has no concept of generic functions.

```
fn foo<T: Debug>(x: T) {
    println!("{}", x);
}

// ..is duplicated into..

fn foo_u32(x: u32) { ... }
fn foo_f32(x: f32) { ... }
fn foo_str(x: &str) { ... }
```

Polymorphisation

Polymorphisation attempts to detect when monomorphisation is unnecessary and therefore reduce redundant duplication of functions, closures and generators.

As a result, less LLVM IR will be generated, reducing compilation time spent in LLVM.

Polymorphisation

This project implements an initial polymorphisation analysis and the surrounding infrastructure for polymorphic LLVM IR generation in rustc.

In the initial implementation, the polymorphisation analysis detects unused generic parameters.

Unused generic parameters are common in closures, which inherit the parameters of the parent item.

```rust
fn parse_value<V>(
    &mut self,
    visitor: V,
) -> Result<V::Value>
where
    V: de::Visitor<'de>,
{
    let peek = match self.parse_whitespace()? {
        // ...
    };

    let value = match peek {
        b'n' => {
            self.eat_char();
            self.parse_ident(b"ull")?;
            visitor.visit_unit()
        }
        // ...
    };

    match value {
        Ok(value) => Ok(value),
        Err(err) => Err(err.fix_position(
            |code| self.error(code))),
    }
}
```

Queries

Compilers are traditionally implemented in passes, where source code is processed multiple times. Each pass takes the result of the previous pass and performs an analysis, such as lexical analysis, semantic analysis or optimisation.

rustc is implemented with a demand-driven architecture, utilizing a query system, which is better suited to integration into development environments and in assisting code completion engines, an expectation of modern programming languages.

Types and substitutions

rustc represents types with a `ty::Ty` type, a recursive type which forms a tree and allows for representation of arbitrarily complex types.

Generic functions and parameters are represented by `ty::Ty` which contain `ty::Param` types, each with an index corresponding to the generic parameters in scope.

`SubstsRef<'tcx>` types represent a list of types which will be substituted for the `ty::Param` types in a type, by indexing into the `SubstsRef<'tcx>`.

MIR

MIR, or “middle intermediate representation”, is a simplified version of Rust, better suited to flow-sensitive analyses.

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 a terminator, which can have multiple successors.

```
fn main() -> () {
  let mut _0: ();
  let _2: bool;
  let mut _3: &mut HashSet<i32>;
  let _4: bool;
  let mut _5: &mut HashSet<i32>;
  let _6: bool;
  let mut _7: &mut HashSet<i32>;

  scope 1 {
    debug set => _1;
  }

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

  // ...
}
```

Monomorphisation

rustc performs monomorphisation to determine which items are generated as LLVM IR. Monomorphisation collects “mono items”, anything which would result in a function or global in the generated LLVM IR.

By walking the HIR (an earlier intermediate representation), non-generic syntactic items are added to a set. For each of those items, the MIR is traversed, and neighbours are discovered, which are also added to the set. By starting with non-generic items, every mono items in the set will be fully monomorphic.

Polymorphisation

A polymorphisation query is introduced, which takes the identifier of a function, closure or generator, and returns a bitset representing whether a parameter in scope is used.

To determine which parameters are used, the analysis traverses the MIR of the item. Whenever a type is encountered, the type is traversed.

If the type contains a type parameter, then the index it contains is set in the final bitset.

To support all uses of generic parameters in Rust code, the analysis has additional handling of closures, generators, constants, predicates, and lifetime parameters.

Polymorphisation

During monomorphisation, function-like mono items are polymorphised.

Results of the polymorphisation analysis are used to modify the `SubstsRef<'tcx>` for the item, replacing unused parameters with the identity parameter.

This results in fewer unique mono items in the set.

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

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

// without polymorphisation, three mono items..

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

// with polymorphisation, one mono item..

foo::<A, u32>
```

Polymorphisation

In addition, modifications to polymorphise the generation of call instructions are required to avoid linkage errors.

However, rustc's code generation infrastructure assumes that types will always be monomorphic, so significant debugging is required to identify where additional changes are necessary.

Evaluation

Compiler performance is measured on 40 benchmarks and compared against a build without this project's contributions. Benchmarks are compiled in debug and release configurations, each 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
- *patched incremental*: an incremental build starting with complete cache and modified source directory

Evaluation

Benchmarking is performed on a Linux host with a AMD Ryzen 5 3600 6-Core Processor, with 64GB of RAM and ASLR disabled. *perf* is used to collect the CPU clock, clock cycles, page faults, memory usage, instructions executed, task clock and wall time during benchmark execution.

Evaluation

Most benchmarks had little performance impact as a result of this project's changes.

However, benchmarks with heavy use of generics had 3-5% compilation time improvements, resulting in up to 11 second savings.

Thanks!

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.