

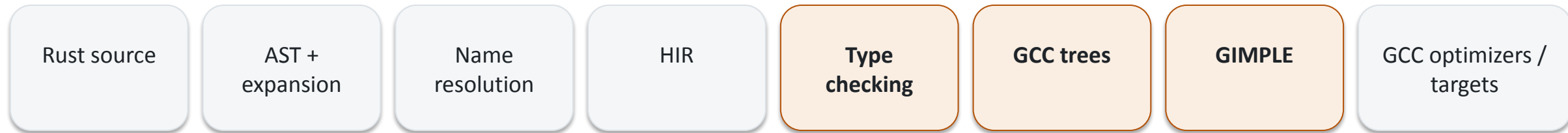
gccrs: State of the Compiler

core, language coverage, GCC integration, and the path toward Rust-for-Linux

Linux Plumbers Conference 2026 — Philip Herron

gccrs in one picture

An independent Rust frontend integrated directly into GCC



- Not rustc with a GCC backend: parsing, HIR, type checking and Rust semantics live in gccrs.
- The backend produces native GCC trees and GIMPLE, so normal GCC optimization and target machinery can take over.

Same Rust, different view inside GCC

A small GIMPLE example: same source semantics, different information reaches GCC

```
pub struct Point { x: i32, y: i32 }  
pub fn point_sum(p: Point) -> i32 { p.x + p.y }
```

gccrs — typed frontend lowering

```
i32 point_sum (const struct Point p)  
  _1 = p.x;  
  _2 = p.y;  
  D.5694 = .ADD_OVERFLOW (_1, _2);  
  ...
```

rustc_codegen_gcc — rustc backend interface

```
int point_sum (int param0, int param1)  
  D.4235 = .ADD_OVERFLOW  
    (param0, param1);  
  MEM[(int *)&binopResult] =  
    REALPART_EXPR <D.4235>;  
  ...
```

Architectural difference: gccrs keeps Rust-level type and source structure visible deeper into GCC.

Both optimize this example to essentially the same machine code — this is not a performance claim. GIMPLE trimmed for readability.

What gccrs can do today

Substantial implementations exist across the language, type system and backend

Language & types

Generics • traits • associated types • GATs •
APIT/RPIT • closures/Fn traits • const generics •
specialization

Semantics

Autoderef/autoref • projections • auto
traits/negative impls • unsizing/coercions • trait
objects • const evaluation

Compiler/library contract

no_core • lang items • intrinsics • rust-call •
macros/decl_macro • SIMD/platform intrinsics •
LLVM→GCC builtin adapters

Backend

Monomorphization • GCC record/vector types •
vtables • DST layout • closure codegen • Drop

Current milestone: Rust 1.49 core

The difficult work is now concentrated in semantic interactions, not basic syntax

Parse / expand — complete

Resolve / HIR — complete

Typecheck — ACTIVE BUGFIXING

Intrinsics — substantial

core codegen — substantial

Consume as crate — substantial

Current bug families

Projection / normalization

Trait-bound satisfaction

Method candidate selection

Inference / substitution interactions

Closure / Fn interactions

Intrinsic coverage

Diagnostic count is a poor progress metric: macro expansion and failed bindings amplify a much smaller set of root causes.

core has driven gccrs forward

Each blocker has exposed another layer of the Rust compiler/library contract

Closures / Fn

implemented / exercised

Const generics

implemented / exercised

Un sizing / DSTs

implemented / exercised

dyn dispatch

implemented / exercised

APIT + RPIT

implemented / exercised

GATs + projections

implemented / exercised

Auto traits + negative impls

implemented / exercised

SIMD + platform intrinsic s

implemented / exercised

Rust 1.49 core requires more than stable Rust 1.49

The library exposes a wider compiler contract than the public stable language

98

feature-gate names in core 1.49

- Min const generics — stabilized 1.51
- GATs — stabilized 1.65
- Associated type bounds — stabilized 1.79
- Inline const — stabilized 1.79
- `min_specialization` / `decl_macro` / `extern_types` — still unstable
- `lang items`, `rustc_*` attributes, `rust-call` and `intrinsic`s — compiler-internal contracts

Independent implementation means reproducing the compiler/library contract as well as the stable source language.

After core: make crates reusable

Current cross-crate support works for small cases, but the metadata model needs to evolve

Today

Export AST as source text

Read metadata payload

Reparsing + resolve again

Lower imported HIR

Revisit imported items in type checking

✓ Small two-crate examples can compile/link/run

Next architecture

Durable crate identity

Exported definitions + visibility

Generic parameters and predicates

Types / projections / impl information

Macro definitions + hygiene

Reliable semantic reuse without duplicate work

Goal: a dependable cross-crate semantic contract

Rust-for-Linux: build on what already exists

A kernel failure is not the same thing as a missing language feature

Existing foundation

Generics / traits

GAT machinery

Const generics

APIT / RPIT

Closures

Unsizeing + dynamic dispatch

FFI + asm frontend

GCC code generation

`#[used(compiler)]`

Concrete gaps to close

GAT associated-type where clauses

Modern arbitrary_self_types / Receiver

`#[derive(CoercePointee)]`

Explicit ``_`` const inference

CFI integration

Asm backend completeness

Cross-crate macros + proc-macro environment

Drop / borrow completeness

Modern sysroot + kernel build integration

Where the wider community can help

The project now has several parallel workstreams that do not require owning the whole type checker

Core semantic bugs

Focused reproducers for projections, method lookup, bounds and inference.

Intrinsics + GCC backend

Atomics, volatile operations, target features, SIMD adapters, ABI and codegen validation.

Metadata + crates

Design and implement a durable cross-crate semantic interface; macro export/import and hygiene.

Rust-for-Linux gap reduction

Reduce real kernel failures; classify missing feature vs bug vs infrastructure; validate modern sysroot needs.

GCC / kernel integration

Language hooks, plugins where required, CFI, build flags, debug/BTF/tooling and Kbuild integration.

Testing + conformance

Execution tests, multi-crate tests, invalid-program tests, optimized-code and debug-info comparisons.

The critical path is no longer one monolithic compiler task: there are meaningful, bounded areas where new contributors and sponsors can accelerate progress.

Roadmap

Turn the current libcore integration work into a usable multi-crate GCC Rust toolchain

NOW

Finish core typechecking
Close root semantic bugs
Expand intrinsic coverage

NEXT

Whole-core codegen
Redesign metadata
Reliable cross-crate macros

THEN

Compile + consume core
Larger crate graphs
Modern sysroot validation

TARGET

Close concrete RfL gaps
Kernel/GCC integration
Real-world deployment

gccrs already contains a substantial Rust frontend. The next gains come from finishing integration, strengthening crate boundaries, and widening the contributor base.

Thank You

Questions, discussion, and contributions are very welcome.

gccrs on GitHub

github.com/Rust-GCC/gccrs

gccrs Zulip

gcc-rust.zulipchat.com

Email

herron.philip@gmail.com

Backup: why does this take so long?

Rust ergonomics move substantial semantic complexity into the compiler

`x.foo()`

receiver inference

autoderef

receiver adjustments

candidate lookup

generic substitution

predicate satisfaction

projection normalization

coercion

result inference

The remaining bugs are increasingly interactions between already-implemented mechanisms.

That is why libcore is such a valuable integration workload.

Backup: what a debugger sees

Same point_sum example, built at -O0 -g and inspected with readelf --debug-dump=info



The same architectural distinction is visible in debug information.

rustc_codegen_gcc debug-info support is still work in progress. gccrs also has gaps: match-pattern bindings are currently missing from its DWARF.