Rust MIR-Level Range Analysis

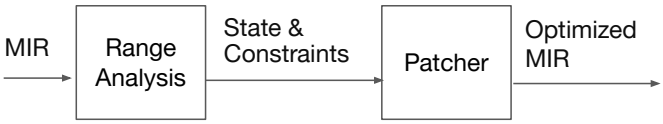
Introduction

Range analysis computes lower and upper bounds on the values that variables may take. These bounds enable optimizations such as dead-code and redundant-code elimination.

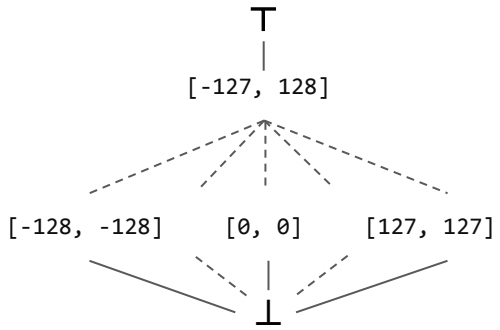
```
fn foo(x: u8) {
  if x <= 10 {
      match x {
          0 => "zero",
          1 => "one",
          20 => "twenty",   // Unreachable
          100 => "hundred", // Unreachable
          _ => "other",
      }
  };
}
```

Design

Our pass is organized as a two-stage pipeline. First, we perform a **path-sensitive**, intraprocedural **dataflow** analysis. Then we use a patching module to remove unreachable `match` arms, prune `assert` statements, and perform constant propagation.



The domain of our dataflow analysis is `Top`, `Bottom`, and `Range`. To guarantee termination on cyclic control flow, each `Range` also carries a join counter. The values within range differ depending on type of the variable. For example, the lattice for `i8` is:



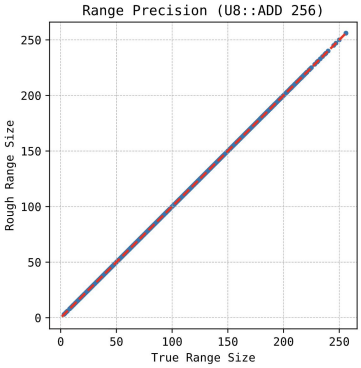
The transfer function acts on unary, binary, and casting operations. We use Rust's constant evaluator to determine overflow and wrapping behavior.

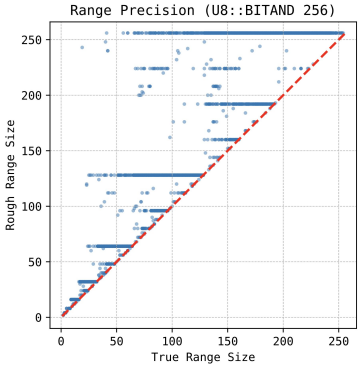
`Cast`	`IntToInt`, `FloatToInt`
`UnOp`	`Not`, `Neg`
`BinOp`	`Add`, `Sub`, `Mul`, `Div`, `Rem`, `BitXor`, `BitAnd`, `BitOr`, `Shl`, `Shr`, `Eq`, `Lt`, `Le`, `Ne`, `Ge`, `Gt`

We implement **path sensitivity** by keeping track of path constraints at each binary and unary operation. We also use an alias map to keep track of copy assignments. When we refine the range for one alias, we propagate the refinement back to its source and other aliases, so later computations still benefit from earlier branch information.

Evaluation

We employ random fuzzing to evaluate our binary operation analyses by verifying soundness and computing the relative size of the output range.





We compared a compiler built from upstream Rust against a compiler built from our branch. We benchmarked with the `compile-benchmarks` in `rustc-perf`. The overall performance impact of our analysis falls largely within the natural variability of compilation workloads.

Metric	Range	Mean
Instructions	[−1.01%, +0.16%]	+0.01%
Task Clock	[−46.78%, +92.06%]	+1.69%
Context Switches	[−68.42%, +87.50%]	-0.36%
Branch Misses	[−30.37%, +7.45%]	-0.39%
Cycles	[−7.09%, +10.13%]	-0.16%

Rust MIR-Level Range Analysis Writeup

Chloe Qiao & Alex Wang CSCI 1951Q [Fall 2025]

https://github.com/qiaochloe/rust-range-analysis

Introduction

Rust's mid-level intermediate representation (MIR) is used for flow-sensitive analyses, including borrow-checking, optimization, and code generation. The MIR lies between the the AST and the low-level codegen backend. Making optimizations at the MIR-layer has two main advantages:

1. The LLVM backend does not need to re-optimize each monomorphized instance, improving compile time.
2. Non-LLVM backends (e.g., Cranelift) benefit from MIR-level optimizations they would not perform themselves, improving runtime performance.

Our project adds a path-sensitive, intraprocedural MIR pass for **range analysis**. Range analysis computes lower and upper bounds on the values that variables may take. These bounds enable optimizations such as dead-code and redundant-code elimination.

Our implementation analyzes casts, unary operations, and binary operations. We use the resulting ranges to:

1. Remove unreachable `match` arms,
2. Prune `assert` statements whose conditions are provably true or false, and
3. Perform constant propagation when a value's range is a singleton.

In the example below, type information implies that `x as u16` is in `[0, 1]` and `y as u16` is in `[0, 255]`. Therefore, `z` lies in `[0, 256]`, and the `assert` is always true and can be removed.

```
fn foo(x: bool, y: u8) {
  let z = x as u16 * y as u16;  // [0, 256]
  assert!(z < 257);             // always true
}
```

Our implementation is also **path-sensitive**, which means that it tracks different program states for each execution path. In the next example, `x as u8` is in `[0, 255]`. It is furthered refined by the `if` statement to be in the range `[0, 10]`.

```
fn bar(x: u8) {
  if x <= 10 {
      match x {
          0 => "zero",
          1 => "one",
          20 => "twenty",   // Unreachable
          100 => "hundred", // Unreachable
          _ => "other",
      }
  };
}
```

Implementation

Overview

We implement range analysis as a dataflow pass using Rust's existing `Analysis` framework. The abstract domain is a `State<RangeLattice>`, where:

- `Bottom` denotes uninitialized values
- `Range(lo, hi, signed)` denotes a closed interval of integer values plus a signedness bit
- `Top` denotes an unknown value

To guarantee termination on cyclic control flow, each `Range` also carries a join counter. After a bounded number of joins (our `JOIN_LIMIT`), we conservatively widen the value to `Top`. This simple widening ensures convergence at the cost of precision in deep or complex loops.

At the start of the analysis, we initialize all function arguments with **type-based ranges** (e.g., `u8` is `[0, 255]`, `bool` is `[0, 1]`). We treat non-integral types as `Top`. The dataflow framework then propagates these ranges forward through statements and terminators. We update the state primarily at:

1. Assignments from binary operations
2. Assignments from unary operations
3. Casts to an integer type

After iterating to a fixed point, we run a separate patching pass on the MIR that uses final ranges to perform redundant code elimination on `switchInt` and `assert` terminators.

Calculating Ranges

Instead of re-implementing integer semantics, we reuse Rust's constant-evaluation interpreter (`InterpCx` with `DummyMachine`) for arithmetic. For each operation, we approximate the output range from the input ranges, using a small set of representative endpoint combinations and asking the interpreter to evaluate those cases.

One challenge was handling overflows. In debug builds, integer overflow triggers a panic; in release builds, Rust uses two's complement wrapping semantics. We decided to model the release mode behavior. We treat overflowing operators as producing a wrapped result plus an overflow flag, and our analysis tracks both a range for the value and a range for the overflow bit.

When an operation cannot be modeled soundly (e.g., unchecked arithmetic or cases where a divisor range may include zero), we conservatively return `Top`.

Path-Sensitivity

Base dataflow over MIR is path-insensitive. To recover some path sensitivity, we maintain a small path-constraint map that records how boolean temporaries were computed. We track constraints of the form:

1. `x op c`, where `x` is a numeric variable, `op` is a comparison, and `c` is an integer constant
2. `!x`, where `x` is a boolean variable

Before every assignment, we check whether we can record a `path_constraint` on a binary or unary operation. Later, when we encounter a `switchInt` on that boolean, we consult the constraint and refine ranges per outgoing edge.

Consider the code snippet below.

```
fn baz(x: u8) {
  if x > 5 {
    assert!(x > 3);
  }
}
```

On the edge where `x` is true, we intersect `x`'s current range with the values that satisfy `x > 5`. On the edge where `x` is false, we intersect with the values that satisfy the negation, `x <= 5`. Inside the true branch, the analysis can then prove `x > 3` is always true.

One challenge with this approach is handling copies. Constraints are often attached to temporaries rather than the original variable.

We decided to maintain a simple alias map for copy assignments. When we refine the range for one alias, we propagate the refinement back to its source and other aliases, so later computations still benefit from earlier branch information.

Patching the MIR

Once the dataflow analysis has converged, we use a `ResultsVisitor` that inspects each statement and terminator using the final ranges. We apply three main optimizations:

1. **Constant folding**. When an assignment produces a boolean or integer with a singleton range, we replace the rvalue with an equivalent constant, effectively doing local constant propagation.
2. **Pruning `assert` terminators.** For `assert!(cond)`, if the abstract value of `cond` is always true, we replace the `assert` with a `goto` to the success block. If it is always false, we replace it with `unreachable`.
3. **Simplifying `switchInt` terminators** If the discriminant's range is a singleton, we replace the `switchInt` with a `goto` to the unique target. Otherwise, we drop arms whose pattern values lie outside the discriminant's range and, when possible, simplify the target table.

All transformations are applied via an `MirPatch` that rewrites the MIR body in place.

Limitations and Tradeoffs

Our analysis makes several simplifications:

1. **Intraprocedural only**. We do not propagate information across function boundaries; return values are treated as `Top`. This prevents us from, for example, eliminating array bounds checks hidden inside library functions such as `Index::index`.
2. **Type coverage**. We only track ranges for integer and boolean types. Other types are conservatively treated as `Top`.
3. **Widening via join limits.** The fixed `JOIN_LIMIT`-based widening strategy ensures termination but can lose precision in deeply nested or highly branching code.
4. **Restricted path sensitivity**. We only model comparisons against constants and simple `!` negations. Correlations between variables (e.g., `x > y`) and more complex boolean structure are not exploited.

Evaluation

We evaluate our range analysis along three axes:

1. **Operator-level correctness and precision.** We check that the abstract transfer functions for integer operators are sound and reasonably precise by comparing them against the ground truth computed by exhaustive or randomized enumeration.
2. **Qualitative effect on generated MIR.** We inspect MIR dumps before and after enabling the pass to confirm that it performs the intended simplifications without introducing obviously buggy transformations.
3. **End-to-end impact on compiler performance.** Because this is an analysis pass in the middle of the compiler pipeline, we measure compilation time on a small suite of representative Rust crates to ensure that the analysis does not introduce a significant slowdown.

Operator Correctness

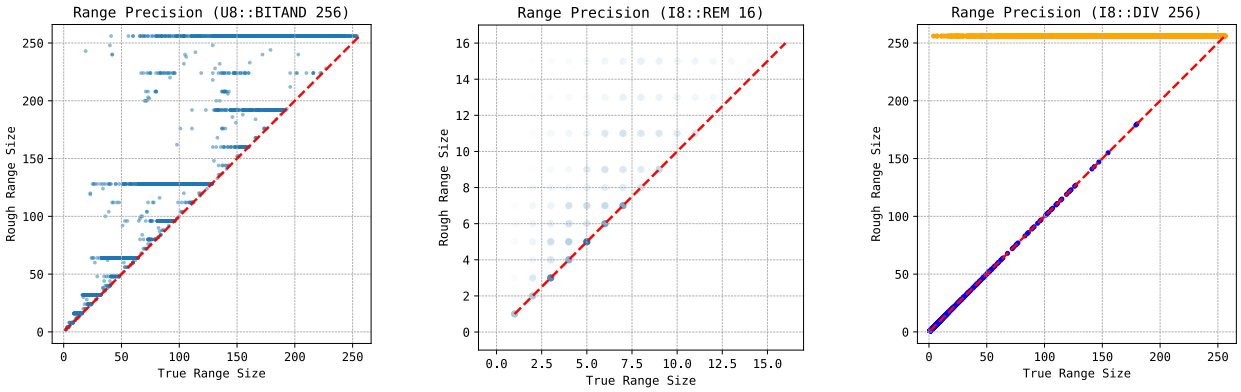
The behavior of an operator can vary depending on type size, type signedness, overflow semantics, and certain error cases for signed division and modulo. To cover these edge cases, since Rust does not provide a scalable testing framework, we employ a probabilistic fuzzing framework in Python to test both soundness and precision.

	u8 ∈ [0, 255] i8 ∈ [-128, 127]			*u8 ∈ [0, 15] i8 ∈ [-8, 7]*		
Operator	**True Size**	**Rough Size**	**Increase (%)**	**True Size**	**Rough Size**	**Increase (%)**
U8::ADD	205.235	205.235	0.000	11.636	11.636	0.000
U8::SUB	205.011	205.011	0.000	173.399	173.399	0.000
U8::MUL	255.479	255.968	0.192	80.985	80.985	0.000
U8::DIV	8.400	9.089	8.204	4.554	33.917	644.698
U8::REM	123.188	170.162	38.132	7.868	11.159	41.838
U8::BITAND	121.069	170.339	40.696	8.104	10.716	32.225
U8::BITOR	121.496	170.920	40.680	8.065	10.662	32.205
U8::BITXOR	203.638	219.383	7.732	12.818	13.718	7.029
U8::EQ	1.669	1.669	0.000	1.709	1.709	0.000
U8::NE	1.669	1.669	0.000	1.709	1.709	0.000
U8::LT	1.666	1.666	0.000	1.668	1.668	0.000
U8::LE	1.666	1.666	0.000	1.667	1.667	0.000
U8::GT	1.666	1.666	0.000	1.667	1.667	0.000
U8::GE	1.666	1.666	0.000	1.668	1.668	0.000
I8::ADD	200.142	200.142	0.000	10.936	10.936	0.000
I8::SUB	200.149	200.149	0.000	10.936	10.936	0.000
I8::MUL	255.531	255.909	0.148	35.503	35.503	0.000
I8::DIV	90.157	133.098	47.629	7.270	142.517	1860.213
I8::REM	85.909	128.183	49.208	5.303	8.230	55.208
I8::EQ	1.672	1.672	0.000	1.705	1.705	0.000
I8::NE	1.672	1.672	0.000	1.705	1.705	0.000

I8::LT	1.669	1.669	0.000	1.659	1.659	0.000
I8::LE	1.670	1.670	0.000	1.661	1.661	0.000
I8::GT	1.670	1.670	0.000	1.661	1.661	0.000
I8::GE	1.669	1.669	0.000	1.659	1.659	0.000

The framework tests all behaviors except for type size. For each operator, it generates $N = 50{,}000$ pairs of random ranges covering either the entire input space (blue header) or a non-overflowing region (green header), computes maximally-restrictive bounds on the output range (true range), and compares with a Python mirror of our range analysis algorithms (rough range).

Soundness is confirmed if the true range is contained within the rough range. The average sizes of the true and rough ranges are listed in the table above along with the percent size increase from true to rough. The size comparison is also plotted below.



Many of the simpler operators align perfectly. Our algorithms have some difficulty with bitwise operators since the lower bits contain so much uncertainty. We compare upper bits, generating streaks around powers of two. Our remainder algorithm also struggles since it is very primitive, only accounting for signedness on the dividend and upper limit on the divisor. Darkness on the plot above corresponds to point frequency. Division, especially on the smaller range, performs the worst since any divisor range containing zero is sent to `Top`, as denoted by the orange points. A more sophisticated algorithm could ensure proper error checks are in place and safely assume that division by zero, even if allowed by ranges, will not occur.

Effect on Generated MIR

We next consider how the analysis affects the compiler's intermediate representation. Using Rust's testing framework, we collect the MIR before and after enabling the pass for a set of small programs. In particular, we check that `assert` statements and `switchInt` statements are removed. Our tests are in `tests/mir-opt/range-analysis/`.

End-to-End Compilation Time

All experiments were conducted on a Cloudflare-hosted virtual machine running Ubuntu 24.04 (x86-64). The VM is backed by an Intel Xeon D-1548 processor (8 physical cores, 16 hardware threads) with a base frequency of 2.0 GHz.

To quantify the cost of the analysis, we compared a stage-2 compiler built from upstream Rust (commit 7c2c3c0) against a stage-2 compiler built from our branch with the analysis pass enabled. We benchmarked with the `compile-benchmarks` in `rustc-perf`.

Metric	Range	Mean
Instructions	[−1.01%, +0.16%]	+0.01%
Task Clock	[−46.78%, +92.06%]	+1.69%
Faults	[−15.55%, +9.65%]	+0.05%
Context Switches	[−68.42%, +87.50%]	-0.36%
Cache Misses	[−7.97%, +23.52%]	+0.21%
Branch Misses	[−30.37%, +7.45%]	-0.39%
Max-RSS	[−3.53%, +4.95%]	+0.24%
Wall Time	[−29.79%, +58.93%]	+1.39%
Cycles	[−7.09%, +10.13%]	-0.16%

Overall, the performance impact of our analysis is small and largely within the natural variability of compilation workloads. That said, the analysis does introduce a measurable overhead, suggesting that realizing its full value may require pairing it with more aggressive downstream optimizations.