Dataflow Analysis on Rust MIR

Andres Rios
ariossta@andrew.cmu.edu

Abstract

This work presents a tool[1] for performing dataflow analysis on Rust programs. It works by leveraging Rust's Mid-level Intermediate Representation (MIR), which provides simplified statements and control flow. We give an overview of the implementation, describe the API and show an example of the tool in practice.

1 Introduction

Dataflow analysis is a powerful tool used in optimizers and static analysis tools to reason about programs. Tools like Soot (Vallée-Rai et al., 1999) enable users to, among other things, perform dataflow analysis in Java programs, and similar tools exist for other languages. However, no such tool exists (to this author's knowledge) that enables a user to easily perform data flow analysis on Rust programs without having to implement the underlying algorithms themselves.

This work implements a dataflow analysis tool on top of Rust's MIR so that a user only needs to specify a lattice and flow functions to be able to perform dataflow analysis on Rust programs.

2 Related Work

The algorithm implemented in this work is Kildall's algorithm (Kildall, 1973). It works by mantaining a worklist of program nodes yet to be analyzed (initialized with the entry node of the program) and a mapping from nodes to dataflow information on node entry. On each iteration, a node is taken off the worklist, the entry dataflow information is modified based on the node's statement(s), and then that information is propagated to the input of the node's successors, as specified by the CFG of the program. The nodes which have their input modified are then added to the worklist.

The MIR was originally proposed in Rust's RFC 1211 (Matsakis, 2015). The MIR (short for Mid-level Intermediate Representation) is a simplified representation of a Rust program. Each function is represented as a set of basic blocks, where each basic block consists of a sequence of statements and a terminator. The statements we are interested in are assignment statements, where an lvalue is assigned to an rvalue. Rvalues cannot be arbitrarily complex expressions - they are at most the result of a binary operator. Terminators are how the control flow is represented. There are several types of terminators (goto, switch, function calls, etc.). An example of a Rust program and its simplified MIR is shown in figure 1.

3 Implementation Overview

3.1 Internal Details

The analysis itself is implemented as a compiler callback. After the MIR is generated, compiling is stopped and the analysis is performed.

The CFG nodes on which the analysis operates are the MIR's basic blocks, not statements. This is not leaked to the public; users need only propagate information through individual statements. Analysing whole basic blocks at a time enables us to keep track of which variables are equal to others, so that the collected dataflow information can be more precise. The reason this is useful is because the MIR always assigns variables to a temporary before using them in a conditional expression. Any information we gain about the temporary from the result of the conditional is also gained about the actual variable, but for that we need to be able to reason that both variables

[1] code available at https://github.com/Arios16/dataflow-rs

```
// Rust source
fn abs(x: i32) -> i32{
    if x < 0 {
        return -x;
    } else {
        return x;
    }
}

// MIR
fn abs(_1: i32) -> i32 {
    let mut _0: i32;
    let mut _2: bool;
    let mut _3: i32;
    let mut _4: i32;
    bb0: {
        _3 = _1;
        _2 = Lt(move _3, const 0i32);
        switchInt(move _2) ->
            [false: bb2, otherwise: bb1];
    }
    bb1: {
        _4 = _1;
        _0 = Neg(move _4);
        goto -> bb3;
    }
    bb2: {
        _0 = _1;
        goto -> bb3;
    }
    bb3: {
        return;
    }
}
```

Figure 1: Example of a function and its (simplified) MIR counterpart

are equal when the conditional is executed. Temporaries are not assigned more than once per basic block, so reasoning about them is easier within a basic block.

As figure 1 shows, `switch` statements do not directly operate on an rvalue producing a boolean, but rather, assign the boolean to a variable and then switch on that variable. This makes it harder to propagate conditional information. A common pattern is for the boolean to be produced within the same basic block (as is the case in figure 1), followed by a possible `Not` statement. The analysis can correctly propagate conditional information in such cases. However, in some cases the boolean value inside the `switch` statement can be generated from multiple places (e.g. multiple blocks assign a boolean to a variable and then use a `goto` to jump to a block that does a `switch` on that variable). In that case, it is difficult to determine which information must be propagated depending on the result of the `switch`, as the condition being evaluated could be one of many. In this case, the current implementation loses the opportunity to refine the dataflow information in the `switch`, simply treating it as a `goto` with multiple destinations. This is sound, but not as precise as it could be.

3.2 Public API

The most important part of the public API is the `Lattice` trait, shown in figure 2. Traits are similar to interfaces in other languages, specifying a set of functions that functions have to implement to satisfy the trait. The lattice trait requires the type implementing it to be able to specify the top and bottom elements, how to join two lattice elements, and how to flow information through assignments and branches. It also demands a method to deal with function calls, but interprocedural analysis is not yet supported. This method exists purely so that the lattice can assign a top element to the appropriate variable, if necessary.

Though lattices can take any form, a very common pattern is to have a lattice that maps local variables to a simpler lattice, as is the case with the canonical sign analysis or interval analysis. For this use case, the API offers the `SimpleLattice` trait, shown in figure 3. Once `SimpleLattice` is implemented for a type `T`, `Lattice` is automatically implemented for a hashmap mapping local variables to `T` values. This eases the burden of implementation for users with this simpler use case.

The last piece of the public API is the `run` function, which receives two arguments: a string `target`, specifying the path to the Rust source file on which to run the analysis, and a function `F` that takes a statement and a lattice element as input. The function `F` will then be called for every statement in the MIR along with the dataflow information that was gathered about that statement. The user can use this `F` function to report errors or for other purposes.

4 Example: Sign Analysis

An example using this tool to perform sign analysis can be found with the linked code. The

```rust
pub trait Lattice: PartialEq + Eq + Sized + Clone + Debug {
    fn bot(decls: &IndexVec<Local, LocalDecl>) -> Self;
    fn top(decls: &IndexVec<Local, LocalDecl>) -> Self;
    fn join(op1: &Self, op2: &Self) -> Self;
    fn flow_assign(
        &self,
        local: Local,
        rvalue: &Box<Rvalue>,
        equiv: &mut HashMap<Local, Vec<Local>>,
    ) -> Self;
    fn flow_branch(
        &self,
        rvalue: &Box<Rvalue>,
        equiv: &mut HashMap<Local, Vec<Local>>,
    ) -> (Self, Self);
    fn flow_function_call(
        &self,
        func: &Operand,
        args: &Vec<Operand>,
        destination: &Place
    ) -> Self;
}
```

Figure 2: `Lattice` trait

```rust
pub trait SimpleLattice: PartialEq + Eq + Copy + Debug {
    fn applies(ty: &TyKind) -> bool;
    fn bot() -> Self;
    fn top() -> Self;
    fn join(op1: &Self, op2: &Self) -> Self;
    fn alpha(a: ConstValue) -> Self;
    fn flow_binop(op: &BinOp, arg1: &Self, arg2: &Self)
        -> Self;
    fn flow_unop(op: &UnOp, arg: &Self) -> Self;
    fn flow_cond_true(op: &BinOp, arg1: &Self, arg2: &Self)
        -> (Self, Self);
    fn flow_cond_false(op: &BinOp, arg1: &Self, arg2: &Self)
        -> (Self, Self);
}
```

Figure 3: `SimpleLattice` trait

example uses the `SimpleLattice` API.

The simple lattice definition can be found in figure 4. The different variants of the `enum` represent whether the variable is zero, lower than zero, greater than zero, etc. The `SimpleLattice` trait is implemented for this type. As mentioned before, this gains us an implementation of `Lattice` for the type `HashMap<Local, PreciseSign>`. The analysis is then run using this lattice along with a function that reports an error if a variable that may be negative is cast as an unsigned integer (which causes an underflow).

```rust
pub enum PreciseSign {
    Top,
    Bottom,
    Lower,
    LowerEqual,
    Zero,
    Greater,
    GreaterEqual,
}
```

Figure 4: `PreciseSign` simple lattice definition

```rust
fn test_fn(mut x: i32, arr: &[i32]) -> i32 {
    return arr[x as usize];
}

fn test_fn2(mut x: i32, arr: &[i32]) -> i32 {
    assert!(x<0);
    return arr[x as usize];
}

fn test_fn3(mut x: i32, arr: &[i32]) -> i32 {
    assert!(x>=0);
    return arr[x as usize];
}


fn test_fn4(mut x: i32) -> i32 {
    let arr: Vec<i32> = (0..20).collect();
    let mut z = 2;
    let mut y = 2;
    while x < 0 {
        x += 1;
        y = z;
        z -= 1;
    }
    return arr[y as usize];
}
```

Figure 5: Functions being analyzed

```
Analysing function: "test_fn"
Possible error at example.rs:3:16:
3:26. Value being cast as unsigned
integer may be lower than 0.

Analysing function: "test_fn2"
Error at example.rs:8:16: 8:26.
Value lower than 0 is being cast
as unsigned integer.

Analysing function: "test_fn3"

Analysing function: "test_fn4"
Possible error at example.rs:26:16:
26:26. Value being cast as unsigned
integer may be lower than 0.
```

Figure 6: Output of the analysis

The functions being analyzed can be found in figure 5, while the output of the analysis can be found in figure 6.

In the first example, `x` may be anything, and so it may be lower than 0 when it is cast as `usize`. This is reported by the analysis.

In the second example, the `assert` statement makes sure that if the program execution reaches the cast, then `x` is guaranteed to be negative, and so the analysis reports a definite error.

In the third example, the `assert` statement ensures that `x` is non-negative if the cast is reached. Thus, the analysis finds no errors.

The last example is slightly more complex. Initially, `y` (the variable that is later cast as `usize`) is positive, and it remains positive for the first iterations of the loop. However, if we iterate enough times then it may become negative. Because the number of iterations is unknown (it depends on the parameter `x`), then we may or may not have an error. The analysis understands this control flow construct, and so reports that there is a possible error.

5 Conclusion and Future Work

A working dataflow analysis tool is presented that can be used to relatively easily perform analysis on Rust programs. However, there are some limitations on this work:

- The analysis is strictly intra-procedural. Implementing iter-procedural analysis is left as future work.
- The analysis can't propagate path-sensitive information for `if` statements with complex conditionals. This is due to the difficulties described in section 3.1. Future research is required to determine how to correctly propagate path-sensitive information in these cases.

References

Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA. ACM.

Nicholas Matsakis. 2015. Mid-level intermediate representation rfc. Available at https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press.