Implementation of Post-Dominators Analysis for the Rust Compiler

IOANNIS MELIDONIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE BACHELOR OF SCIENCE DEGREE IN COMPUTER SCIENCE

UNIVERSITY OF CRETE
SCHOOL OF SCIENCES & ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

SUPERVISOR: POLYVIOS PRATIKAKIS

2022

Abstract

Increased demand for reliable, safe and efficient software has led to modern programming languages aiming to incorporate these principles as part of their core, using Static Code Analysis. One of these languages is Rust. Static Code Analysis is the analysis of source code during compilation instead of a run-time environment. This study shows the process of integrating Post-Dominators Analysis into the Rust Compiler and provides a better understanding of the compiler and how it handles static analysis. Post-Dominators Analysis is a Backward Control-Flow Static Analysis providing information to other analyzers and optimizers. It finds for each point of execution of a function, which other parts is certain that they will run in the future.

For the evaluation, the Rust Compiler source code is used as the code base for the measurements, because it is one of the largest software written in Rust and thus it is a representative sample. The source code was compiled by the *rustc* version that includes the Post-Dominators Analysis. All metrics were extracted by using a function inside the compiler that took the role of the consumer for the analysis. The evaluation is focused on what types of graphs the compiler is processing in general and how the structure of analyzed graphs affects the result of Post-Dominators Analysis.

Contents

List of Figures

List of Code Snippets

1 | Introduction

With the extensive usage of software across many different domains, more emphasis is given to code quality. It is important to provide programs without unexpected crashes, bugs or poor performance. Also, identifying and solving security weaknesses is crucial, especially for software designed for the web.

Thus, besides correct results, developers want to ensure their code does not have any security vulnerabilities and identify parts where bugs may rise and solve them during development, producing more reliable software. Also, when programming for systems with limited resources or time-critical programs, often the code needs optimization before producing the final program, to secure it can actually run and accomplish its tasks. A good tool to confront these challenges is Static Code Analysis.

This thesis describes the process of adding *Post-Dominators Analysis* to the Rust Compiler. Post-Dominators Analysis is a static analysis that calculates for every part of a program, if that part runs, which other parts will definitely run too before the program exits. The result, for example, can be used by *Control-Dependence Analysis*, *Implicit Flow Handling* [2], *loop-invariant* code motion [3] and testing *code-coverage* for test sets [4].

The compiler already includes many static analyses running in each compilation, ensuring the reliability and robustness of rust software. The rust development community constantly tries to improve both performance and safety by adding more optimizations and analyses to the compiler. However, Post-Dominators Analysis was not yet implemented.

1.1 Static Code Analysis and Rust Background

Static Code Analysis is a methodology of collecting information about a program without executing it. Examples of what a static code checker could detect are unused or uninitialized variables, unreachable parts of the code and potential memory leaks. Even though static analysis is used more often in recent software development, it originates back in the late 70s. In 1978, Stephen C. Johnson developed *Lint* [5], one of the first static checkers, establishing the term *linting* for any static analysis. It was a tool used for analyzing the code of C programs to detect certain bugs. Lint was an external tool, independent of the C compiler. Programs in C could still compile normally, without requiring to get analyzed by it.

Throughout the years, static analyzers have evolved. More tools have been released, like *Splint* and *PMD*, and plugins for IDEs like *CodeRush* for Visual Studio. Also, compilers integrated various static analyses. Although these programs can help in the development cycle, the developers decide whether they want to use them, it is not enforced by any language standard. Moreover, when a language allows the programmer to write unsafe code freely, it is harder for static analyzers to find precise results.

Modern programming languages are trying to reduce unsafe code, allowing static analyzers to increase their precision and report less false-positives. They usually have strict syntax and type rules and thus it is easier for the analyzers to identify and report errors. A good example of such a language is *Rust*, which is the language chosen in this thesis.

In 2006, Graydon Hoare began working on a programming language called Rust focused on providing a better alternative to systems programming and producing software with guaranteed memory safety [6]. Rust combines features from many different languages including *OCaml*, *Haskell* and *Ruby* [7] and on top of that, it builds a memory model that checks every memory access, during compilation. In addition, the compiler does many analyses both for identifying bugs and optimizing the code. Rust is an example of a language that tries to integrate static analysis as part of its core and not as an option to developers.

1.2 Overview of the thesis

The following chapters contain a detailed description of the process along with information about the analysis and Rust. Chapter 2 describes the main features of Rust, the architecture of the compiler and explains the components used for the implementation. It also contains a high-level description of Post-Dominators. In chapter 3, a detailed description of the implementation is given, including the algorithms for finding Post-Dominators and Immediate Post-Dominators. Chapter 4 shows the results of the evaluation process and includes findings about the analysis of this thesis and the general representation of data that the compiler analyzes. The final chapter summarizes the implementation and highlights some improvements that can be applied to the algorithms.

2 | Background

2.1 Why Rust?

Rust is a programming language, designed by Graydon Hoare and developed as an open-source project. Hoare started Rust as a side-project in 2006, Mozilla sponsored his work in 2009 and it was officially released in 2010 [8], while its first stable version was announced in May of 2015 [9]. Rust provides both low-level control and high-level tools that ensure the correctness of a program. Its main focus is safety and performance.

One of the main features of Rust is the memory safety it provides without using any runtime monitoring or garbage collection, which would add overhead during the execution of a program. Rust manages memory through an *Ownership* mechanism, enforcing every value to have a single owner, which means that only one variable can hold the actual value. After an owner goes out-of-scope the value is dropped automatically, without the programmer having to explicitly free the memory.

Assigning an owner to another variable, passes the ownership to the second one, making the first invalid to use from that point on. Although a value can be owned by a single variable, other variables can *borrow* that value by holding a reference to it, with the restriction that either a single mutable or multiple immutable references can exist simultaneously. When a variable that holds a borrowed value goes out-of-scope, the actual value is not dropped because the variable is not its owner. This way, Rust prevents double-free errors.

This feature is also useful for thread-safety. A common error in parallel programming is a *data-race*, when two threads are accessing the same memory and one of them mutates it. Rust, using the Ownership Model, ensures that no data-race is possible, as a value can either be shared with many read-only references or one mutable. Other similar features are available through both Rust's Standard Library and external libraries like *Rayon*.

Rust also comes with a build tool called *Cargo*. Using Cargo, a programmer is able to handle bigger projects with many different local or external dependencies. A developer just writes which dependencies are needed and Cargo is responsible to find, download if necessary and compile them. It uses a configuration file, called *Cargo.toml*, to determine how to compile a project, which dependencies are used and other useful information like name, version and authors of the project. It can also publish to the Rust community's crate registry called *crates.io*, a public collection of many crates to use in any project.

The Rust book [10] has an extensive list of all its features along with examples for further explanation. In addition, the compiler has very descriptive warning and error messages to inform a programmer for any mistake and also suggests solutions to the problem, providing a more beginner-friendly developing environment. Moreover, its large community provides more features and guides constantly, resulting Rust to become a great choice for developers. According to *Stack Overflow*'s survey in 2021 [11], Rust has the first place as the most loved programming language. Also, from the same survey, it has the fifth place as a language that developers who are not using it yet, want to program with in the future.

2.2 The Rust Compiler

Rustc is the compiler of Rust. It produces the binary code, either as an executable or as a library. It is designed to need only the *root* file of a crate to compile it. A crate in Rust is a single compilation unit consisted of one or more files linked together using the *mod* keyword. The root of a crate is the file that following its *mod* declarations, the compiler can find all other source files it needs to compile the crate. Unlike other languages like C/C++, the programmer doesn't have to manually compile every file and then link them together.

The compiler of Rust is also written in Rust. This means it needs bootstrapping to get compiled. It uses an older version of the compiler to compile the newer version and the process is broken into stages [12]. Stage 0 is the older compiler, stage 1 is the output of compiling the new source code with the stage 0 compiler and stage 2 is the output of compiling stage 1 with itself.

2.2.1 Compiler Architecture

Compiling a source file in Rust, as shown in Figure 2.1, goes through various steps before outputting machine code. The first step is to analyze the text into a manageable format for the compiler to further analyze the code. Source code goes through a *lexing* phase to produce the tokens later used by the parser to construct the *Abstract Syntax Tree* (AST). *Rustc* follows the top-down parsing method to construct the AST. After this phase, all macros are expanded, the AST is validated, all names are resolved and some early lints have run [12].

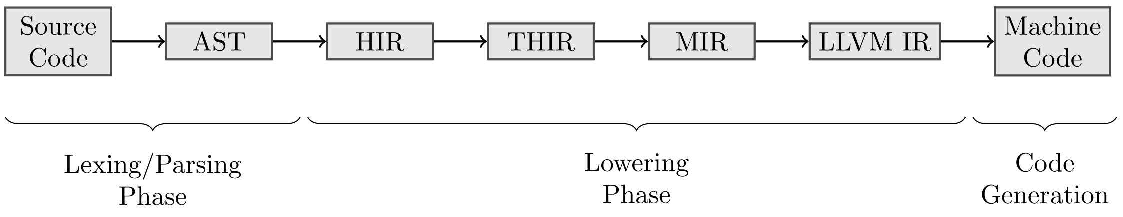


Figure 2.1: Rust Compiler Architecture

Next step is the *lowering* phase, in which the AST is transformed into some intermediate representations (IRs), easier for *rustc* to process. Each lowering desugars the previous IR further to bring it in a more suitable form for the analyses that will take place.

Initially, the AST is converted into the *High-Level Intermediate Representation* (HIR), removing any information not useful in following analyses. This IR is mainly operating around types. Type inference, trait solving and type checking occur during this phase [12], producing the *Typed High-Level Intermediate Representation* (THIR).

Then, *rustc* lowers THIR to *Mid-Level Intermediate Representation* (MIR). It is constructed as a Control-Flow Graph (CFG), allowing the compiler to analyze the flow of a program. Many optimizations use MIR because it is less complex than all previous IRs and use the CFG for Control-Flow and Data-Flow Analyses. Also, the Borrow Checker uses MIR to analyze source code for memory-safety. When all optimizations and checks have run, *rustc* has to go through one last step before starting *code generation*.

The compiler uses LLVM as its back-end to produce the final machine code, which requires a special kind of IR called *LLVM Intermediate Representation* (LLVM IR). *Rustc* takes the optimized MIR, converts it to the LLVM IR and passes it to LLVM. With this IR, LLVM does more optimizations and finally outputs the machine code.

2.2.2 MIR Structure

As mentioned before, MIR is a Control-Flow Graph, making it convenient for analyses that depend on the flow of a program. Instead of generating the MIR for the whole program, each function has its own. *Rustc* represents the nodes of the graph as *Basic Blocks*. A Basic Block is a collection of sequential statements without intermediate branching. The last statement contains the successors of the node and is called *Terminator*. The successors are the Basic Blocks that the control flow can continue to. Depending on the kind, a terminator can have zero or more successors.

2.2.3 Query System

One of the concerns of *rustc* is to reduce compiling time for a crate. Rust achieves faster compilations with *incremental compilation*. Instead of rebuilding a crate from scratch, it compiles only those parts that are affected by a modification. Rust approaches incremental compilation with the usage of a *query system*. Internally, various components of the compiler are querified and can be called through their appropriate method in *TyCtxt*, the main struct of *rustc* [12].

The idea behind queries is when they're invoked for the first time, the compiler will execute them and cache their result. If nothing changes, each subsequent invocation will return the cached result, making them *memoized*. Then, after compilation finishes, *rustc* can store the results of queries together with the dependency between them, and in later compilation, retrieve and use each result that is not affected by the changes. Most queries work like this, but some of them have a modifier to their definition instructing the compiler to recompute the result in every compilation.

Once a query is involved and the result is not already computed, *rustc* needs to call the function that calculates it. These functions are called *providers*, which are passed to the *TyCtxt* through the struct *Providers*. Because providers are defined per-crate, each crate that has one, needs to have a function *provide*, which is responsible for adding any provider defined in that crate to the Providers struct.

2.2.4 Built-in Analyses of *rustc*

Many analyses are already implemented for *rustc*. The compiler has built-in tools, called *lints*, that use these analyses both to optimize the final binary and to make sure the source code is correct. Most of the lints will generate warning or error messages [13] to inform programmers if any potential bug was found. Some of them are:

Liveness: Liveness Analysis, finds for each program point, which variables are live and which are dead. A variable is called *live* at a point, if it may be used later is any execution path, starting from that point onward, without being overwritten. Otherwise, it is called *dead*. *Rustc* uses this analysis to warn for dead assignments or variables that were not used in a function, whether they are locals, arguments or captured variables. To do that, it uses an implementation for data-flow analysis over the AST. There is another implementation which operates over the MIR and is mainly used in the transformation of generators into state machines.

Dominators: Dominators Analysis is the opposite of the analysis presented in this thesis. For each node of a CFG, it calculates the set of nodes that are guaranteed to have executed before it. The algorithm used is based on Loukas Georgiadis dissertation [14] and Lengauer-Tarjan paper [15]. The result is useful during the Borrow Checking phase and Coverage Analysis. It is also used in the code generation phase of the compiler when it needs to find the non-SSA (Static Single-Assignment) locals.

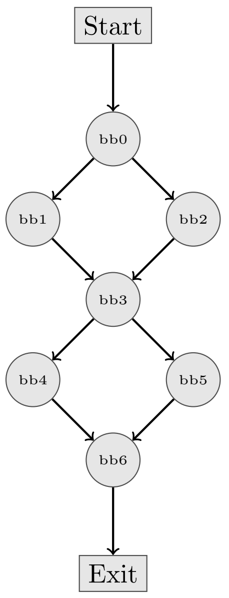
Exhaustiveness and Reachability: Rust uses patterns extensively. The most common place where a pattern can be found is a *match* expression. *Rustc* uses Exhaustiveness and Reachability Check to ensure when pattern-matching occurs, every possible pattern is considered and there is not any value that cannot be matched, and also that each pattern is reachable. Because patterns are checked top-to-bottom, if a lower pattern tries to match a value already matched from a higher one, it is unreachable. This analysis is not used only for match expressions found in the source code. Any other expression that translates to an MIR containing a match expression, like *if let* and *while let*, is also analyzed [12].

Rust uses more analyses including *Dead-Code*, *Polymorphization* and *Capture* analysis and has a wide list of lints, like *const_item_mutation*, *large_assignments* and *unconditional_recursion*, contributing to improve the overall quality and correctness of the source code and improving the performance of the generated binary code.

2.3 Post-Dominators

Given a Control-Flow Graph $G = (N,\ E,\ EXIT)$, with nodes N, edges E and a single $EXIT$ node, we say that a node $x \in N$ post-dominates a node $y \in N$ if every path from y to $EXIT$ contains x. Node x is the *post-dominator* of y and this relation is expressed as: x pdom y. The post-dominators of a node n, are all those nodes that if the execution goes through n, it is guaranteed that they will be executed too.

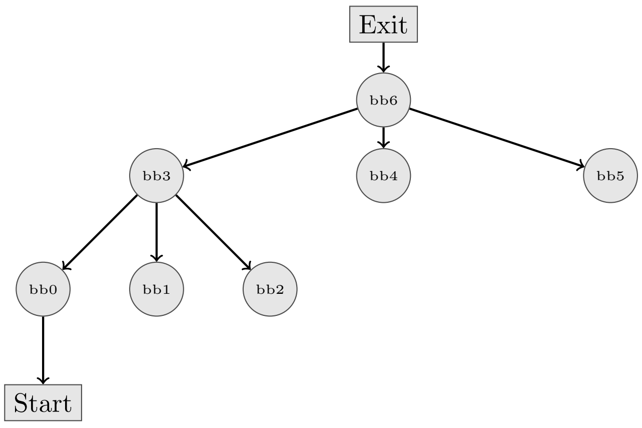
Post-dominator though carries more information than is needed. It is often more convenient to find the *immediate* post-dominators. Every node $n \neq EXIT$ has a unique immediate post-dominator which is the post-dominator closest to n that is not n. For $EXIT$, we assume that its immediate post-dominator is itself. If we represent the immediate post-dominators as a tree with $EXIT$ being the *root* and each vertex has as parent its immediate post-dominator, then by traversing the tree from a node n back to the *root*, we can find all the post-dominators of n. That tree is called *post-dominator tree.*



(a) Control-Flow Graph

n	PDOM(n)	IPDOM(n)
Start	{Start, bb0, bb3, bb6, Exit}	{bb0}
bb0	{bb0, bb3, bb6, Exit}	{bb3}
bb1	{bb1, bb3, bb6, Exit}	{bb3}
bb2	{bb2, bb3, bb6, Exit}	{bb3}
bb3	{bb3, bb6, Exit}	{bb6}
bb4	{bb4, bb6, Exit}	{bb6}
bb5	{bb5, bb6, Exit}	{bb6}
bb6	{bb6, Exit}	{Exit}
Exit	{Exit}	{Exit}

(b) Result of Post-Dominators Analysis



(c) Post-Dominator Tree

Figure 2.2: Post-Dominators Analysis Example

Figure 2.2 shows an example of the Post-Dominators Analysis. Given the graph in 2.2a, the result computed is shown in 2.2b. Each node of the graph is a Basic Block (bb). For node $bb3$ there are only two possible paths from $bb3$ to $Exit$:

- $bb3 \rightarrow bb4 \rightarrow bb6 \rightarrow Exit$
- $bb3 \rightarrow bb5 \rightarrow bb6 \rightarrow Exit$

In both sequences, three nodes are common: $\{bb3, bb6, Exit\}$, so these are the post-dominators of $bb3$. Node $bb4$ is not a post-dominator because there is another path that reaches $Exit$, starting from $bb3$, without passing through $bb4$: $bb3 \rightarrow bb5 \rightarrow bb6 \rightarrow Exit$. Also, its immediate post-dominator is $bb6$, because it is closer than $Exit$. For this graph, the post-dominator tree is shown in Figure 2.2c. Taking $bb6$ as an example, it has $bb3$, $bb4$ and $bb5$ as children, because all three of them have it as their immediate post-dominator. Furthermore, to reach $Exit$ from node $bb1$, following the parent relation, the nodes that will be traversed are $\{bb1,\ bb3,\ bb6,\ Exit\}$ which are the post-dominators of $bb1$ according to the table in Figure 2.2b.

3 | Design and Implementation

3.1 Algorithm for Post-Dominators Analysis

Post-Dominators Analysis is a Backward-Must Data Flow Analysis. The facts it collects are transferred from the *Exit* node of the Control Flow Graph to the predecessors of each node and the final solution for each node holds those facts that must be true in every execution path. For each node n, the answer to this data-flow problem is a set of all those nodes that post-dominate n.

This problem can be solved with the backward data-flow algorithm. As *meet* operation, the *intersection* is suitable because, in each merge-point, only those facts that were true for each successor should be kept. This can be summed up in the following equation:

$$PDOM(n) = \begin{cases} \{n\} & n = exit \\ \{n\} \cup \{\cap_{s \in successors(n)} PDOM(s)\} & n \neq exit \end{cases}$$

As shown in Code Snippet 3.1, with the assumption that each node other than *Exit* has as post-dominator every node of the graph and the *Exit* post-dominates only itself, iterating over the graph's nodes (excluding *Exit*) passes the information from the successors of a node to the node itself. This way, the information is carried backward.

```
// N: Nodes of the graph
fn post_dominators(graph(N, Exit)) {
    pdom[Exit] = { Exit }
    pdom[N - Exit] = { All nodes N }

    change = true
    while change {
        change = false
        for each node n except exit {
            tmp = n + { intersection of all pdom[successors(n)] }

            if tmp != pdom[n] {
                pdom[n] = tmp
                change = true
            }
        }
    }
}
```

Code Snippet 3.1: Post-Dominators Algorithm [1]

3.2 Immediate Post-Dominators from Post-Dominators

The algorithm described in Code Snippet 3.1, calculates all the post-dominators of each node, but this information is superfluous. By calculating the immediate post-dominators, we can find the closest node that post-dominates, and represent the solution as a tree called *post-dominator tree.* Each node of the graph is represented as a node of that tree and its parent is its immediate post-dominator. Traversing that tree from a node n back to its root we can find all its post-dominators.

The algorithm in Code Snippet 3.2 is based on the algorithm for finding the dominator tree [16], but instead of starting from the *Start* node, it starts from the *Exit* and works backwards. First, it's necessary to remove from each set, the *'self'* node. Immediate post-dominator of a node n is never itself, except the *Exit.* Starting with *Exit* node, the node will be removed from each set and then, if a node has an empty set, it denotes that *Exit* is its immediate post-dominator. For every such node, the process is repeated using that node.

```
// N: Nodes of the graph
// pdom: Post-dominators of each node.
fn immediate_post_dominators(graph(N, Exit), pdom) {
    for n in N { pdom[n].remove(n) }

    remaining_queue = { Exit }
    while node = remaining_queue.pop() {
        for n in N {
            pdom[n].remove(node)

            if pdom[n].is_empty() && ipdom[n] not found {
                ipdom[n] = node
                remaining_queue.push(n)
            }
        }
    }
}
```

Code Snippet 3.2: Immediate Post-Dominators Algorithm

3.3 Required Adjustments

3.3.1 Graphs with multiple Exit Nodes

Both algorithms shown in Code Snippets 3.1 and 3.2, are expecting the CFG to have a unique exit node. When *rustc* generates the MIR for a function with multiple return expressions, each one is translated to a *goto* terminator which jumps to a basic block containing a single *return* terminator. However, multi-exit MIRs can be generated too. For example, if it contains a *panic!* macro inside an *if* expression, the graph will have one node for normal return and one for panic. Therefore, both algorithms require modifications to support multi-exit graphs.

Post-Dominators Algorithm will have to take as input a list of exit nodes and initialize *pdom* for every node n as follows:

- if n is one of the exit nodes then $pdom(n) = n$
- else $pdom(n) =$ all nodes.

This way, each exit node will post-dominate only itself and not all other exits. When two branches leading to different exit nodes are merged, it is not possible to know from which one the function will exit, so both will be omitted. The drawback is that some nodes will not have immediate post-dominators later, but the MIR doesn't have to merge all different exits before the analysis. After initialization, inside the loop, each exit node needs to be skipped.

The Immediate Post-Dominator Algorithm has to take into account that an internal node can be empty since it can be the merge point of branches with different exit nodes. During initialization, each exit should have itself as immediate post-dominator and be added to the queue. Next, every node removes itself from its post-dominator set but it needs to check if the set is empty afterwards, signifying they will not have any immediate post-dominator. Those nodes are also pushed to the queue to find for which node they are immediate post-dominators later. Otherwise, during the loop, these merge points will never get pushed to reduce the sets of nodes they post-dominate, resulting empty immediate post-dominators for every predecessor of those points.

Continuing to the reduction step, when node n is popped from the queue and tries to remove itself from the post-dominator set of a node u, the *if* statement inside the *while* loop (Code Snippet 3.2) should make an additional check whether the set is reduced or not. If the set was empty from the start, current algorithm would falsely consider n as immediate post-dominator of u.

3.3.2 Graphs without Exit Node

A function can have an MIR where all Basic Blocks have a successor. For example, if a function contains an endless loop without any breaking point, the graph generated by *rustc* will look like a circle, where all Basic Blocks have an edge to at least one other node. When a Control-Flow Graph does not have an exit node, a solution for Post-Dominators is undefined and the analysis will terminate without finding a result.

3.4 Code Integration

Since the algorithms are presented, next step is to find out how to add this analysis to *rustc*. Some components from *rustc* are needed both to implement the algorithms and make the compiler able to call the analysis and take the results. The two major parts implemented are the Client, the consumer of the analysis, and the Analyzer itself. Before implementing them, the compiler needs to be configured for development as described in the Guide to Rustc Development [12].

3.4.1 The Client

The client is the function responsible for calling Post-Dominators Analysis, taking the result and making some measurements, which then outputs to the console. Because the client resembles a consumer and can get replaced by an actual lint, it is implemented as a Query. Considering that, it needs an entry in the macro containing all other queries and the *provide* function to add the client to the *Providers* struct [12].

To define the client's query, an entry is added to *rustc_queries!* macro with the same name as the client function. If they have different names, the compiler cannot link the query with its provider.

The *TyCtxt* gets the providers from *rustc_driver* which collects them by calling a top-level *provide* function in each crate. That function adds each provider defined in that crate. Thus, the root file of the crate the client belongs to, also requires a *provide* function to call the inner one and eventually add the client to *TyCtxt*.

Rustc has a query called *analysis* that runs analyses for a crate. The client will be called from there to run Post-Dominators Analysis and print the results, after all other analyses. The analyzer then is called for every definition that has an MIR, like functions and constructors.

3.4.2 The Analyzer

Post-Dominators algorithm is integrated following the structure of Dominators Analysis. It is designed in a similar way to ensure the code is close to the design of *rustc* and preserve uniformity across different modules. The client should be able to call the analyzer for every function. Internally, MIR's representation of a function is a struct called *Body*, so a method responsible for calling the analyzer is added to it.

When the analyzer is called, the first step it takes is to find the Exit Node of the graph. *Rustc* uses traits to make the functionality of graphs abstract and then implements them for any type that needs to be a graph. A new trait is introduced to add a method that calculates the set of exit nodes. That trait is then implemented for *Body* to return the set of exit nodes. That method iterates over all basic blocks and adds those without successors to the set. Since a *Body* can be borrowed, the trait is also implemented for reference to *Body*.

The analyzer takes the returned set and checks whether is empty or not. As mentioned in Section 3.3.2, a graph might not have any exit nodes. When the set is empty, the analyzer returns an empty solution along with a flag implying it wasn't able to solve the problem. Otherwise, it continues with the analysis.

4 | Evaluation

The evaluation focuses on providing information for the implementation of Post-Dominators Analysis and also describing the static analysis environment of Rust. In particular, it includes the behavior of the analysis in graphs with multiple exit points and the general structure of graphs that the static analyzers of Rust are processing. The evaluation is performed using the client from Section 3.4.1. The client prints, for each function, the following information to the console during compilation:

- The name of the function
- The number of nodes its MIR has
- How many of these nodes are exit nodes
- The number of nodes without an Immediate Post-Dominator
- The number of unique Immediate Post-Dominators
- The number of unique Immediate Dominators

These metrics are useful to determine the size and complexity of the Control-Flow Graphs for Rust functions and also show how the number of exit nodes affects the solution of Post-Dominators Analysis.

4.1 Data Collection

The *rustc* source code is a large code base of Rust and a representative sample of the language, thus it is used as the code base for the measurements. Using the modified compiler, the source code is recompiled and the next sections include the results found.

To gather the data from the compiler, the stage 2 compiler should be built. To do that, the stage 1 compiler should compile itself. In the data set, the final stage of building, where the stage 2 compiler recompiles everything, is excluded because it produces duplicate results.

During the whole process, the analyzer was called 161846 times and for only three functions the solution was undefined, which are excluded from following measurements.

From the final data set, the names of the function were checked to find out if there were duplicates. Each name has the following format: *crate_name*[*stable_crate_id*]*path*. The id inside the brackets is a hash of the name and some other metadata. None of the functions had the same name with any other, however, when the id from the name was removed, some functions had the same *crate_name* and *path*.

For these functions, it is uncertain whether they are recompiled and the hash changed because of some metadata or they are different functions. So, to avoid duplicate results and increase the accuracy of the evaluation, if they also had the same metrics then only one was kept in the data set. The data set used in the following analyses contains the remaining 148977 functions.

4.2 The Size of Graphs

The bar-plot of Figure 4.1 illustrates the size of the graphs for all analyzed functions. The graphs are divided into six different ranges of nodes. For each range, it shows for how many functions the size of their graph is in that range.

As shown in Figure 4.1, most MIR graphs generated are small, between two and ten nodes. Also, it is observed that a large portion of analyzed functions contain a single node. Constructors and small inline functions often do not have any control-flow resulting one basic block with all their code. Further inspection of generated MIRs showed that rust translates constants into a *Body* with a basic block that returns their associated value.

The node of such graphs is both entry and exit resulting their Immediate Post-Dominator and Dominator to be themselves. So, single-node functions are not providing any meaningful data for further analysis and they are removed from upcoming measurements, reducing the data set's size to 103209.

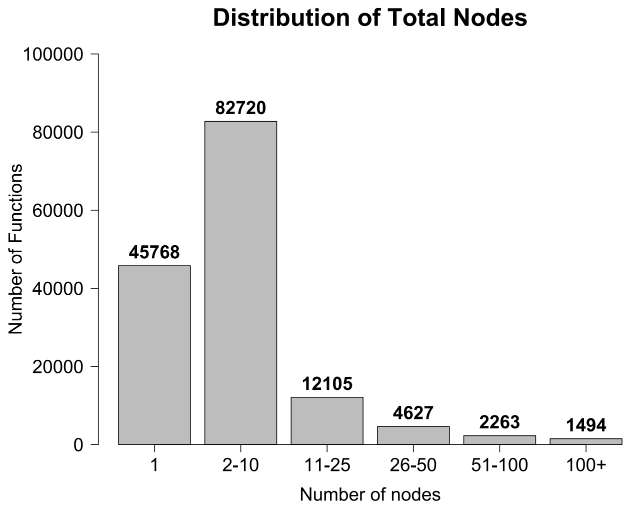


Figure 4.1: Distribution of Total Nodes

4.3 The Complexity of Graphs

The number of unique Immediate Dominators can demonstrate the complexity of graphs from entry to exit. A large graph with few Immediate Dominators indicates the graph has many paths. To analyze this relationship it is better to consider the percentage of nodes that are Dominators in the graph instead of the actual number, as it is more comparable to other percentages. Dominators Analysis is similar to Post-Dominators so it is used to compare the forward with the backward complexity and see if they differ.

The histograms in Figure 4.2 describe how the ranges of percentages are distributed across functions. It shows how often the graph of a function has a specific percentage of Immediate Dominators/Post-Dominators nodes.

Both histograms show that in most functions, the $\sim$ 45% of nodes are Immediate Dominators or Post-Dominators. Also, the distribution around 45% is similar in both histograms. This means that the functions of the code-base have similar forward and backward complexity, in general.

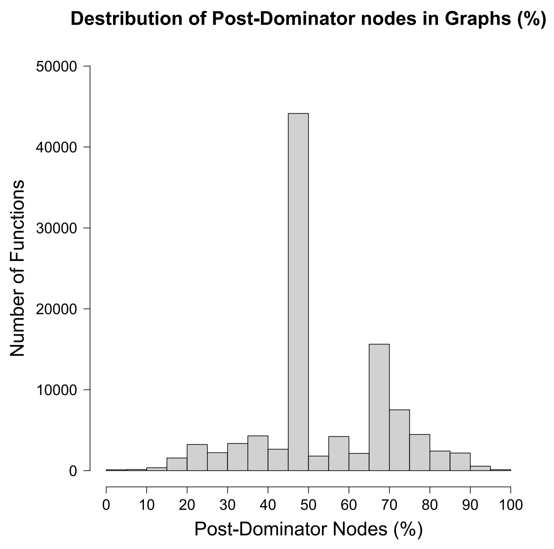
It is observed that more graphs have less than 40% nodes as Post-Dominators compared to Dominators. The main reason behind this is the fact many graphs have nodes without Immediate Post-Dominators, as Section 4.5 will describe, and thus the overall number of Immediate Post-Dominators and therefore the percentages are reduced.

Another information extracted from the same histograms is graphs have higher chance to have more than 45% nodes in their unique sets than having less. Though, for higher percentages, the amount of function decreases. To understand this better, two variables should be taken into account: the size of unique set of Immediate Dominators and Post-Dominators and also the size of the graphs.

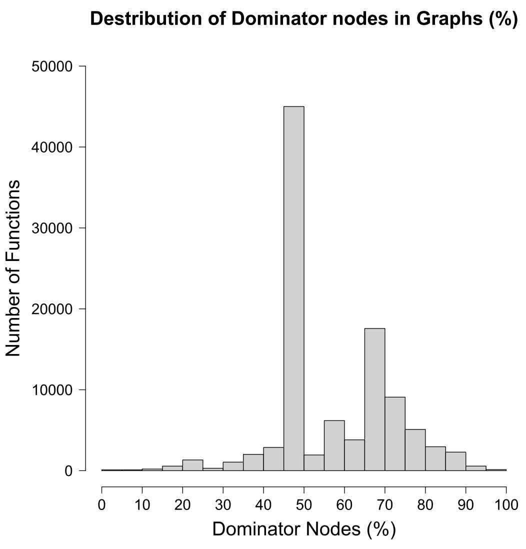
For any given graph with more than one node and a single exit, the set of distinct Immediate Post-Dominators will always be at best one less than the set of nodes. The reason is, the exit node and its direct parent will have as Immediate Post-Dominator the exit node and thus when taking the unique nodes these two will always have a duplicate, reducing the available different nodes by one. For multi-exit graphs, each node that does not have an Immediate Post-Dominators will also reduce the size of the set.

The same relation is true for Immediate Dominators. For each graph with more than one node, the entry and its direct child will have the entry node as Immediate Dominator. So the set of distinct Immediate Post-Dominators is also, at best, less by one than the set of nodes.

In addition, most graphs are small according to Section 4.2, meaning any difference between the size of the graph and the size of the set has greater impact on the ratio than larger graphs, resulting more variation between percentages. So, the percentages are more evenly distributed and less function have higher than 80%.



(a) Post-Dominators



(b) Dominators

Figure 4.2: Percentages of Post-Dominators and Dominators in Graphs

4.4 Exit Nodes and Nodes without Post-Dominators

The bar-plot of Figure 4.3 has the number of exit nodes as ranges and shows for each range, how many functions have that many exit nodes. From the bar-plot, it is visible that most functions have only one exit node. Functions with two exit nodes are $\sim 75\%$ less and very few functions have more than five exits. Thus, it is expected for most graphs to have a small amount of nodes that do not have Immediate Post-Dominators.

The correlation between the size of the graphs and the number of exits is shown in Figure 4.4. The heat map represents each combination of exit node ranges and size of graphs as a box that has a darker color if more functions fall into that combination.

The raw data that were used for constructing the heat map are shown in the table in Figure 4.4b. Because 70% of the functions generate a graph with size between two and ten nodes and they have only one exit, as shown in that table, the frequencies are scaled to avoid a white heat map.

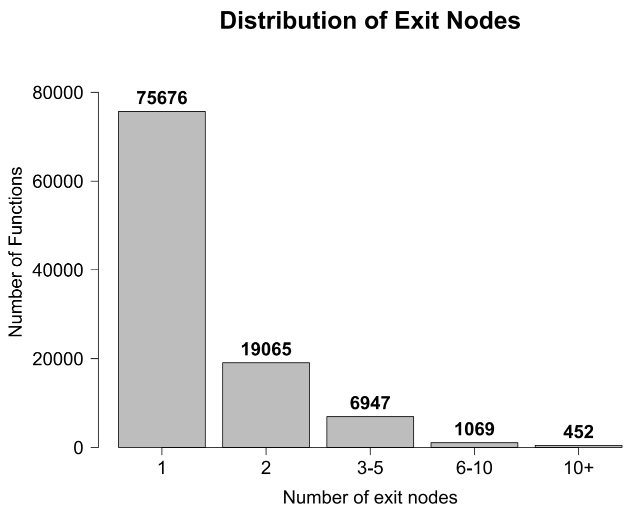
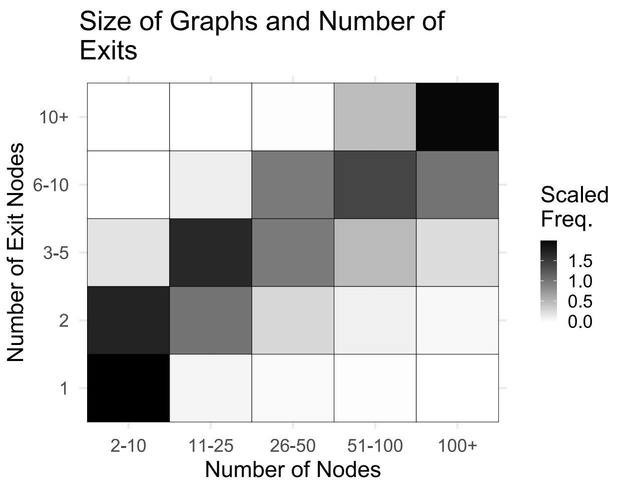


Figure 4.3: Distribution of Exit Nodes



(a) Heatmap

Nodes	Exit Nodes				
	1	2	3-5	6-10	10+
2-10	72051	10301	368	0	0
11-25	2579	6219	3271	36	0
26-50	772	1666	1892	294	3
51-100	177	631	939	430	86
100+	97	248	477	309	363

(b) Raw Data

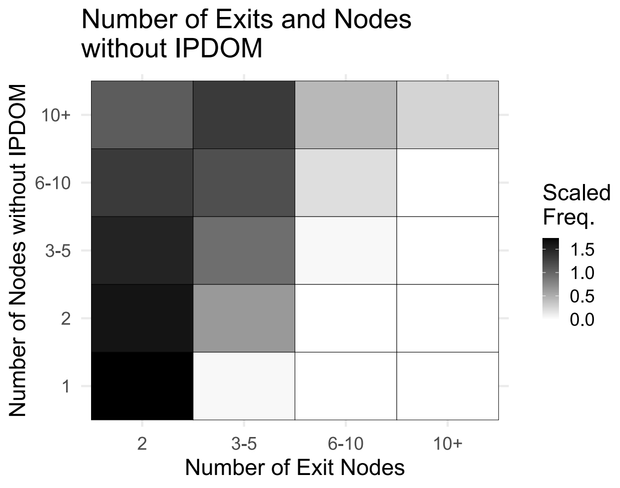
Figure 4.4: Size of Graphs and Number of Exits

In figure 4.4a, all dark boxes are concentrated in a straight ascending line. Therefore, the size of a graph is correlated to the number of exit nodes. Because the line is ascending, the correlation is positive. As the graphs are getting larger, the number of exits tends to increase as well.

As mentioned before, most functions have a single exit node. All nodes of these functions have Immediate Post-Dominators. So, the next plot excludes these graphs and focuses only on graphs with more than two exits. As explained in Section 3.3.1, these functions can have nodes without Immediate Post-Dominators.

Figure 4.5 provides information about whether the number of exit nodes affects the number of nodes that do not have Immediate Post-Dominators. Along with the heat map, a table of the raw data can be found in Figure 4.5b. Each box of the heat map is a combination of exit nodes and nodes without Immediate Post-Dominators and it is darker if more functions have that combination. Again, to avoid a white heat map, the frequencies are scaled.

Analyzing the heat map from Figure 4.5a shows that the number of exit nodes in a graph, when that graph has more than one exit, does not affect the number of nodes without Immediate Post-Dominators. However, graphs with many exit nodes have more nodes without Immediate Post-Dominators. The diversity of coding patterns that lead to multiple-exit graphs, causes the variety of results in the 2-5 range.



(a) Heatmap

Exit Nodes	Nodes without IPDOM				
	1	2	3-5	6-10	10+
2	9655	2761	3208	1522	1919
3-5	186	1071	2005	1358	2327
6-10	1	0	73	230	765
10+	0	0	0	3	449

(b) Raw Data

Figure 4.5: Number of Exits and Nodes without IPDOM

4.5 Information passed to other Analyzers

Post-Dominators Analysis finds a vector with the Immediate Post-Dominator of each node from a given Control-Flow Graph. This result can be used by other analyzers and optimizers, like Control-Dependence Analysis for example. Section 3.3.1 stated that merge-points for branches leading to different exits are assumed to have *None* Immediate Post-Dominators. Thus, when the solution is passed to the consumer, these nodes will not provide any information. In *rustc* source code, 27533 functions had at least one node without Immediate Post-Dominator.

The data set of the analysis consists of 1150735 nodes, if functions with one node are excluded. From that amount, 306826 are nodes without Immediate Post-Dominators. This means, for the large code base of *rustc* source code, any analyzer that would use the result, will be missing $\sim 26\%$ of the information.

5 | Conclusion

In summary, this thesis presents the algorithms to find the Post-Dominators and Immediate Post-Dominators in a Control-Flow Graph. It also shows how these algorithms are integrated into the Rust Compiler along with a basic structure of a consumer for the analysis. Moreover, it gives overview of the compiler and explains its various parts. Finally, we evaluated the implementation using a large code-base and analyzed the results.

The results from the evaluation provided better insight into how Rust handles static code analysis and how the current implementation behaves. Using the source code of *rustc* as a sample code base, the client of Post-Dominators Analysis made some measurements for every function of the compiler. It was found that in a large code base, such as *rustc*, the analyzed graphs are small in general and they tend to have similar forward and backward complexity. Also, as graphs are getting larger, the amount of exit points usually increases. However, the number of exits does not seem to be the main factor that increases the nodes that do not have any immediate Post-Dominator.

5.1 Future Work

The algorithms of Post-Dominators and Immediate Post-Dominators presented in Chapter 3 calculate correct results, however they are not optimized. The algorithm in Code Snippet 3.1 recalculates the post-dominators of each node on every change. This can be optimize with the usage of queue. When the post-dominators set of a node changes, its successors are added to the queue to check in the next iteration if they are modified too and continue to propagate that change. Thus, in each iteration, only the nodes that can be affected by a modification are recalculated instead of the whole graph.

Another potential modification to the algorithm is the way it traverses the nodes. If they are traversed in the order of appearance from the exit to start, it might reach to fixed-point faster, though it has to be measured and find out if it has an impact on the performance of the algorithm.

For the algorithm of Immediate Post-Dominators in Code Snippet 3.2, during the calculation, if the immediate post-dominator of a node is found, it will never change again. So, when it is initially found, if the N set is reduced, those nodes will not be reexamined in every iteration.

As mentioned in Section 3.3.2, graphs that do not have an exit are ignored and the solution is undefined. The evaluation showed that for Rust Compiler source code only three functions fall into this category. However, if this pattern starts to appear more often in the future, a modification to the algorithms should be consider, to find a more generic solution in such graphs to allow a potential optimizer to analyze these functions.

Eventually, the client should be replaced by a real optimizer or analyzer that will benefit from Post-Dominators Analysis. Some suggestions where given in Chapter 1, like Control-Dependence Analysis and loop-invariant code motion.

5.2 Acknowledgements

I would like to thank professor Polyvios Pratikakis for his guidance and assistance throughout my Bachelor's Thesis. Through our discussions, I have learned many things both related to this specific research topic and computer science in general. I am grateful for the opportunity he gave me for collaboration. Also, I would like to thank CARV laboratory of FORTH-ICS, for providing me a space where I was able to work and study for my thesis. Finally, I want to thank all my family members and friends who supported me these years.

References

[1] August David. *Lecture 2: Basic Control Flow Analysis.* 2004. URL: `https://www.cs.princeton.edu/courses/archive/spr04/cos598C/lectures/02-ControlFlow.pdf`. (Accessed on 22 July 2022).

[2] Abhishek Bichhawat. "Post-dominator analysis for precisely handling implicit flows". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* Vol. 2. IEEE. 2015, pp. 787–789.

[3] Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. "Loop quasi-invariant chunk detection". In: *International Symposium on Automated Technology for Verification and Analysis.* Springer. 2017, pp. 91–108.

[4] Hiralal Agrawal. "Dominators, super blocks, and program coverage". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1994, pp. 25–34.

[5] Stephen C. Johnson. *Lint, a C program checker.* Bell Telephone Laboratories Murray Hill, 1978.

[6] *Interview on Rust, a Systems Programming Language Developed by Mozilla.* URL: `https://www.infoq.com/news/2012/08/Interview-Rust`. (Accessed on 3 September 2022).

[7] *The Rust Reference.* URL: `https://doc.rust-lang.org/stable/reference`. (Accessed on 31 August 2022).

[8] Matt Asay. *Rust, not Firefox, is Mozilla's greatest industry contribution.* 2021. URL: `https://www.techrepublic.com/article/rust-not-firefox-is-mozillas-greatest-industry-contribution/`. (Accessed on 7 August 2022).

[9] *Announcing Rust 1.0.* 2015. URL: `https://blog.rust-lang.org/2015/05/15/Rust-1.0.html`. (Accessed on 7 August 2022).

[10] *The rustc book.* URL: `https://doc.rust-lang.org/book`. (Accessed on 25 August 2022).

[11] *Stack Overflow Developer Survey 2021.* URL: `https://insights.stackoverflow.com/survey/2021`. (Accessed on 25 August 2022).

[12] *Guide to Rustc Development.* URL: `https://rustc-dev-guide.rust-lang.org`. (Accessed on 9 August 2022).

[13] *The rustc book.* URL: `https://doc.rust-lang.org/rustc/what-is-rustc.html`. (Accessed on 21 August 2022).

[14] Loukas Georgiadis. *Linear-Time Algorithms for Dominators and Related Problems.* Princeton University, 2005.

[15] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141.

[16] Pingali Keshav. *Dominators, control-dependence and SSA form.* 2019. URL: `https://www.cs.utexas.edu/~pingali/CS380C/2019/lectures/Dominators.pdf`. (Accessed on 30 September 2022).