"ALEXANDRU-IOAN CUZA" UNIVERSITY, IAȘI

FACULTY OF COMPUTER SCIENCE

MASTER'S THESIS

Solidity Optimization using Control Flow Graphs

proposed by

Iacob Sergiu

Session: july, 2022

Scientific Coordinator

Dr. Arusoaie Andrei

”ALEXANDRU-IOAN CUZA” UNIVERSITY, IAȘI

FACULTY OF COMPUTER SCIENCE

Solidity Optimization using Control Flow Graphs

Iacob Sergiu

Session: july, 2022

Coordinator

Dr. Arusoaie Andrei

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Static analysis has been at the core of **optimizing** code for many years now, doubled by its sibling, dynamic analysis. Common programming languages such as C++, Java, Golang etc. generate **bytecode**, by the usage of **compilers**, that is eventually interpreted by various virtual machines, such as LLVM, on various architectures — x86, x86_64, AMD64, arm64 etc.

While the virtual machines that run bytecode and the cpu architectures are shared across programming languages, since they ultimately revolve around the same assembly instructions, each programming language has its own compiler while, of course, interpreted languages such as Python have their own interpreter.

Zooming in on how a compiler works, we find the optimizer, which is a sort of Maslow pyramid level, but in the compiler world. While any compiled code may run just fine without it being optimized, it will more than likely miss on many performance improvements that will greatly speed up the code's run time and resources' usage.

Writing high level code greatly simplifies the development process of software applications. It saves a lot of time, as writing low level code such as assembly often means a great deal of work, but necessary in some situations such as embedded software. Developers cannot always invest the needed time and knowledge to perfectly optimize their code, nor can they make a cpu's frequency higher by writing a line of code. What the compiler can do, instead, is less. Running less instructions, in a better

order, using registers more efficiently and moving less memory blocks around is what gets us the fastest computation. This is ultimately the optimizer's job in a compiler — find opportunities to generate more efficient bytecode.

1.2 Objective

The purpose of this thesis is to research the techniques of applying static analysis on high level code, in order to obtain efficient bytecode, regardless of the virtual machine it will execute on. These techniques are usually backed up by **intermediate representations** of the initial code, which can be either another high level code (with different syntax, but the same semantics), or data structures meant to provide an overview (or in-depth view) over the program execution.

In this thesis, we will see how intermediate representations help, why they're useful, and how **Control Flow Graphs** determine which optimizations can be applied while doing static analysis. The programming language we will focus on is **Solidity** [1].

1.3 Motivation

Solidity is a new programming language on the market, backing up the development of Smart Contracts mostly deployed on the Ethereum blockchain. Since it is a relatively new technology, there is a lot of ongoing work to still build the basis of the eco-system and to enhance the compiler by basic features or common optimizations.

An important aspect is that Solidity is an open-source technology, which facilitates **external contributions**, some of which will be provided by the usage of this thesis. Optimization sits at the heart of the compiler, and as the official documentation states, [3, the optimizer is under heavy development]. This is sufficiently encouraging to research on how Control Flow Graphs, intermediate representations and static analysis techniques could be combined in order to improve the bytecode generated from Solidity.

[1] Solidity is a highly volatile language at this point, therefore it is probable that some of the details presented in this thesis will lose veridicity in the near future. However, the high-level schemas, approaches and structure of the Solidity eco-system will likely remain the same.

1.4 Thesis structure

Bringing external contributions to a technology usually follows a logical timeline, around which this thesis was also structured. Specifically in our case, we'll go through understanding Solidity and its usecases, understanding what a compiler and what a optimizer is, how Solidity's compiler works, what are some of its optimizer's drawbacks and how we should solve these. Finally, we go through integrating our improvements into the actual Solidity codebase.

In the first chapters, we set the knowledge base needed to improve the optimizer. We go through Solidity and the (relatively) new Yul IR, static analysis, control flow graphs (CFG), abstract syntax trees (AST) and the Solidity optimizer structure.

In the latter chapters, we do a deep dive into a few optimization steps and identify edge cases not optimized by these and do a few benchmarks to see how much gas is saved through these improvements. Finally, we conclude with some experiments on Etherscan, where we try to see if the enhancements have an impact on verified smart contracts.

Chapter 2

Contributions

2.1 A way to contribute

The most challenging part was building smart contracts that were not fully optimized by Solidity's compiler. This mostly meant finding situations, edge cases in which the compiler was "fooled" by the high level code, making it impossible for various optimizations to be applied or simply ignored.

The first approach here was to experiment with Solidity and compile smart contracts until such an edge case was found. The challenge here is that it is very difficult and time consuming to analyze the generated bytecode and try to find an optimization opportunity, since assembly instructions are not "human readable".

The second approach was to reverse engineer Solidity's open-source codebase, get familiar with the internal optimizer structure and try constructing edge cases not treated by the optimizer. While this is definitely a good approach, it requires a bit of experience with optimizers and some time to get familiar with Solidity's codebase.

What helped by a great margin was the usage of **YUL Intermediate Representation**, which can be analyzed in its optimized form, since it is the intermediate code used to generate the final bytecode. This way, it was much faster to analyse wether the generated bytecode would be optimal or not.

2.2 Enhancement of optimization steps

Solidity's optimizer has 32 documented optimization steps. One of the most straightforward way to speed up code computation is to run less code, which comes from generating the proper bytecode and from pruning "dead code", i.e. high level code that the user wrote but is unreachable. Enhancements were brought to **UnusedAssignEliminator** and **StructuralSimplifier** that ultimately improved gas consumption for specific scenarios in smart contracts.

In this thesis, several edge cases are presented that tricked the optimizer, and by handling those, the functionality of the whole optimization suite was improved thanks to the cascading optimization aspect [1]. A few other observations are made throughout the thesis, which can easily transform into external contributions to ultimately generate a more efficient smart contract.

2.3 Reducing gas usage of smart contracts

The quality of Solidity's compiler and optimizer is evaluated by the **gas usage** of the smart contract, both deployment on the network and the cost of running a function within the deployed contract. With the optimizations mentioned above, the optimizer **generated code that requires less gas**, which means less ethereum, which means smaller costs for the user.

Multiply any small, seemly insignificant gas improvement by the millions of smart contracts on the Ethereum blockchain and you end up with an overall significant cost reduction in code execution. Therefore, external contributions as these will sum up to bring a real difference in code quality[2].

[1] Improvements to a single optimization step enable other optimization steps (or the same one) to improve bytecode. An example is given in for StructuralSimplifier 5.4.

[2] We will dive deeper into which smart contracts will benefit from these optimizations and how optimization steps give more opportunities to other optimization steps in the latter chapters.

Chapter 3

Solidity

3.1 Overview

Solidity is a relatively new programming language, its first proposal coming in 2014 from the co-founder of Ethereum, Gavin Wood [2]. Its purpose is to serve development of smart contracts to be deployed on various blockchain platforms, especially focusing on the Ethereum blockchain.

Although it's been around for around 8 years now, Solidity has just started to peak interest in the blockchain development community. Dune Analytics reported an increase of 75% in smart contracts deployment in March, 2020, summing up to a total of 2 million smart contracts deployed [4]. Since then, hundreds of thousands of smart contracts have been deployed monthly, with around 370 thousands just in May 2022 3.2.

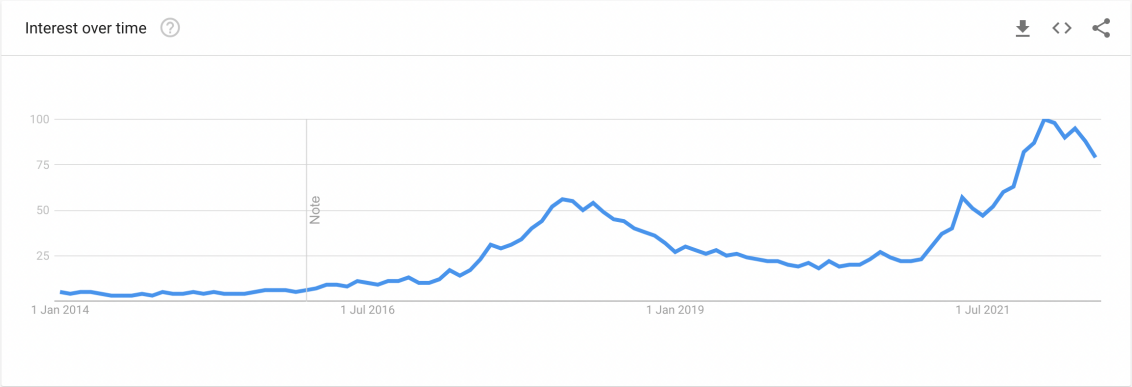

Figure 3.1: Solidity interest from 1st of January, 2014, to 1st of June, 2022, according to Google web searches

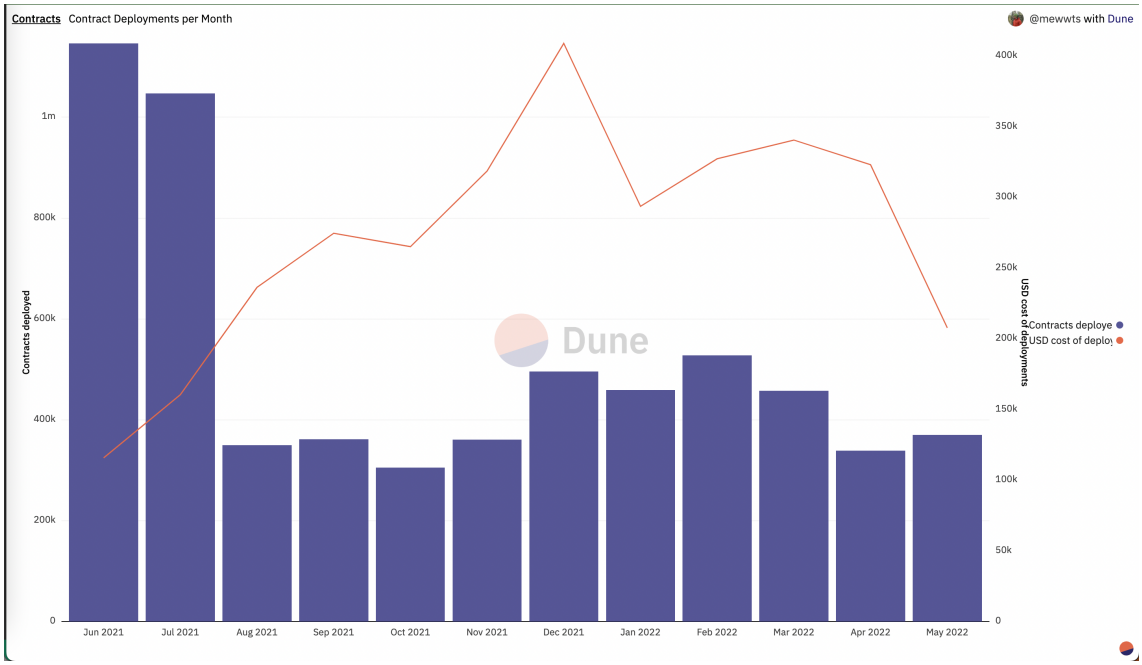

Figure 3.2: Contract deployment per month in the past year, according to Dune Analytics

The main properties that interests this thesis are that this programming language is **statically typed** and is converted into bytecode to run inside the Ethereum Virtual Machine (EVM). Being statically typed helps the compiler with static analysis, making possible the development of an optimization suite that performs both common optimizations and particular optimizations for EVM.

3.2 Solidity's Compiler and Optimizer

High level solidity code is sent to `solc` binary, which stands for "solidity compiler". The user may or may not specify the usage of the optimizer with the `--optimize` flag, which is enabled by default. Upon analyzing the compilation flow 3.3, we notice that there are some optional (yet recommended) steps to follow while compiling, which take advantage of Yul (intermediate code representation for Solidity) and Abstract Syntax Trees, the latter being built for both the initial Solidity code and for the Yul IR. If we choose to optimize the code through Yul, then the `--via-ir` must be specified, which is also enabled by default.

When optimizing through Yul, the compilation flow will actually use **two optimizers**: the "new" one (Yul optimizer) and the "legacy" one (EVM bytecode opti-

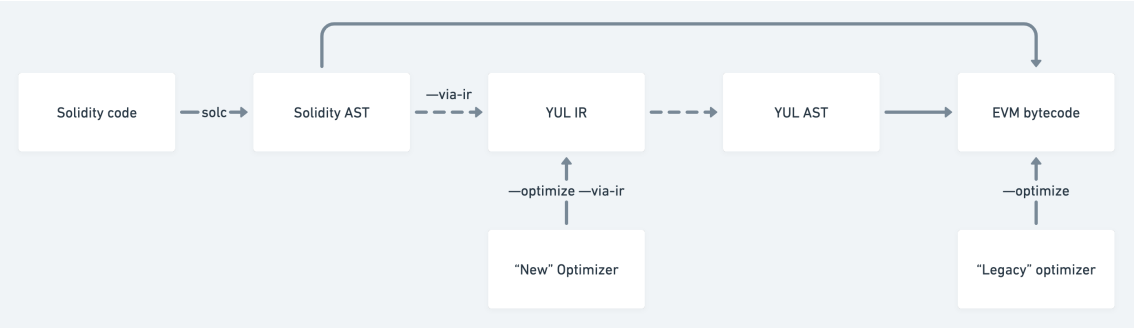

Figure 3.3: Solidity compilation flow

mizer). The reasoning behind this is that bytecode optimization should be as simple and straightforward as possible, this also being an engineering decision made by Solidity's team that they explained in the Solidity Summit presentations, 2022. `JUMP` instructions are especially tricky to handle while optimizing bytecode, making high level, semantic optimization (for example simplifying `for` loops) more difficult.

The advantage of having two optimizers is that it decouples work by a great deal. Low level assembly optimizations are done by the legacy optimizer [1], and more complex optimizations, that require techniques such as data flow analysis, are done directly on the Yul IR, which is later used to generate the efficient bytecode. This also helps to visually inspect the generated Yul IR and making sure that its output is the expected one. Specifically for the team, it means that it makes unit testing and development much easier.

3.3 Intermediate Representations

Abstract Syntax Tree

Abstract Syntax Trees (AST) are often used to do a first analysis of the code instructions by walking the tree. This is often coupled with the Visitor Design Pattern, where the user implements what the `visit` function should do for each type of node. They are often used in compilers to represent the structure of the code, to make isolated code changes and then re-generate code from the AST.

[1] Future work might also consist in unifying EVM and LLVM, bringing all of the work and optimization done to LLVM on Solidity's ground as well.

Figure 3.4: Solidity AST Example. The body of the function is an AST node itself, in which the statements are other AST nodes. Generated using AST Explorer.

On Solidity's territory, ASTs appear in two phases. First, an AST is generated from the unaltered Solidity code, while another AST is later generated from the Yul IR code. The AST nodes provide sufficient granularity to make a structural analysis simple enough, thanks to recursively analysing the children node of a root.

ASTs are very common in static analysis and is often times coupled with a grammar (or dialect) that enforces different node types and node structure in the tree. There are a variety of tools that use ASTs to do code analysis of their own.

Yul IR

Yul, former name Julia, is an intermediate representation for Solidity Code. It has been publicly announced as "production ready" in March, 2022, in version 0.8.13 of Solidity. Since then, the engineering team has been encouraging people to compile smart contracts via the usage of Yul, since the optimization focus has shifted on this pseudocode language.

Yul was designed around four main principles, as described by its official documentation [7]: readability, easy manual inspection of control flow, simple and straightforward generation of bytecode from Yul and whole-program optimization. As compared to bytecode, this IR provides high-level syntax for instructions that introduce jump instructions (`JUMP`, `JUMPDEST`, `JUMPI`), the reasoning behind this being that it is much easier to analyze data flow and control flow in the code.

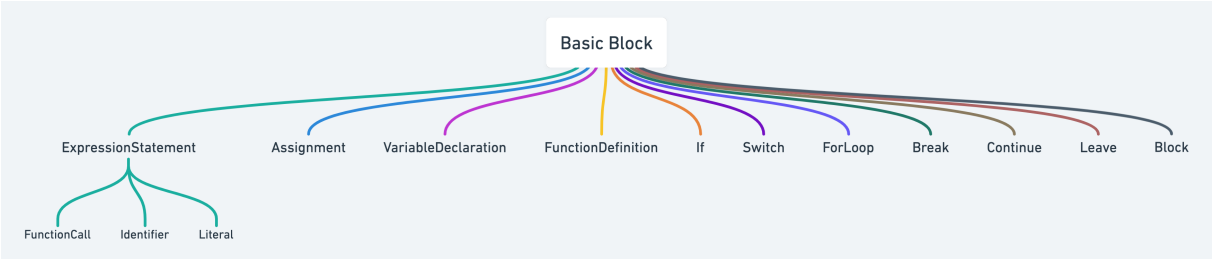

Figure 3.5: Node types in the YUL AST. Node properties vary depending on their nature.

```
{
    function power(base, exponent) -> result
    {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default
        {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

Listing 3.1: Example of Yul code which computes exponentiation recursively

This was designed as an actual programming language, making it possible to generate valid EVM bytecode from Yul code. Of course, it is not recommended to write smart contracts in Yul, but one can do so to get more familiar with the dialect. It also takes on the property of its "parent" code, and is also statically typed as Solidity. This further eases the static analysis done by the compiler.

3.4 Other tools performing static analysis

The compiler has the option of outputting the Solidity AST in JSON format, which means that external tools can be built to alter the AST in any desired way. Since the compiler also supports receiving an AST as input, and generate bytecode from that one, that means that in theory one could build a better optimizer than the official one.

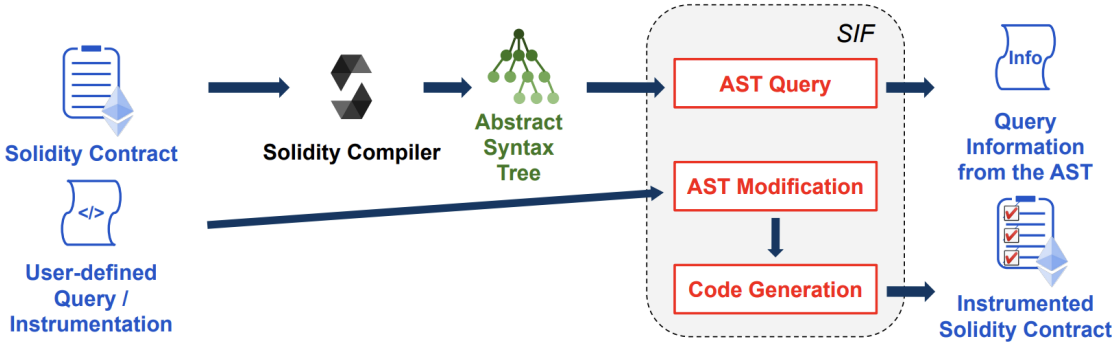

Figure 3.6: SIF workflow, captured from SIF: A Framework for Solidity Contract Instrumentation and Analysis [5]

While that hasn't happened yet, there are a few tools that perform code analysis of their own.

3.4.1 Solidity Instrumentation Framework (SIF)

Built by Chao Peng, Sefa Akca and Ajitha Rajan, from University of Edinburgh, SIF is a tool built in C that gravitates around the Visitor Design Pattern. The main objective is to build an internal C representation of Solidity code, through intermediary data structures, upon which SIF can orchestrate various optimizations / refactoring functions. As the authors describe it, it's a tool [5, to easily and effectively understand, manipulate and analyse Solidity code].

The drawbacks that this toolkit currently has is that it's outdated and that it does not allow for external users to "plug-in" their own intermediate representations of Solidity code. It acts more as a middle-man, allowing users to implement, through the *visit* method, specific optimization / refactoring items that run against isolated pieces of code. The tool does build a Control Flow Graph for the code, the one that this thesis will focus on, but it does not specifically use it for any kind of in-house optimization - it is just available to the user for usage. However, there are issues of Control Flow Graph completeness. ¡- TODO aici paper-ul cu completeness

3.4.2 Slither

Slither is a more abstract, high level static analysis framework, which gives a broader overview over the internals of an Ethereum smart contract. Its main focus is centered around 4 areas: security (vulnerability detection), automated code optimization, code

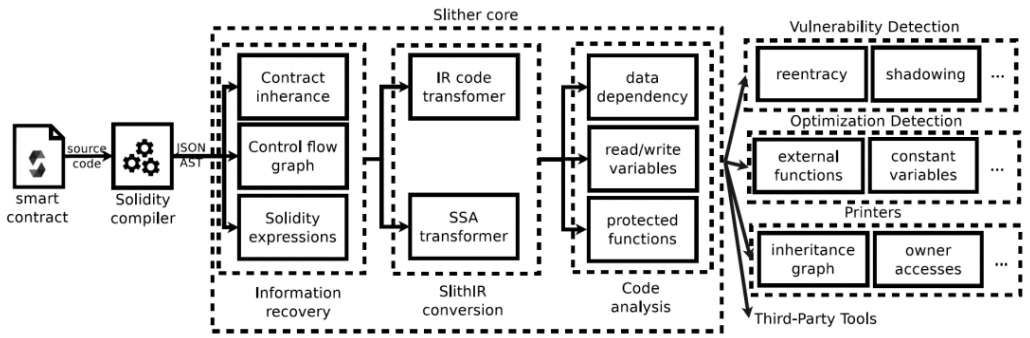

Figure 3.7: Slither Architecture

analysis and assisted code review, as described in Section 4.1 by the author Margherita Renieri [6].

The tool takes advantage of the Abstract Syntax Tree (AST) built by the Solidity Compiler, then passes that through a series of optimization steps: Information Recovery, SlithIR Conversion and Code Analysis.

What's common in both of the tools we've seen is the usage of Abstract Syntax Trees and Control Flow Graphs, as well as building "proprietary" intermediate representation of the input code.

3.4.3 Why focus on the official optimizer

The issue with such tools as SIF or Slither is that they get rapidly deprecated, since Solidity ground is so volatile, and they don't provide better optimization flows than Solidity's compiler does. Furthermore, integration is a bit more difficult, since it requires a separate optimization pipeline to be run through this intermediary tools, which just adds an additional burden to generating efficient bytecode.

What is certain is that the official compiler and optimizer will stay at the core of generating the Solidity bytecode. Therefore, any improvements to that will reflect itself in the next Solidity release - given it passes the team's rigurous reviews, of course.

Chapter 4

Static Analysis

Overview

Static Analysis has been at the heart of inspecting the behaviour of software, without actually executing it, for decades now. It sits at the core of optimizing compilers, but it also gives room for building external tools such as linters. These usually appear in the development pipeline before creating a software release and points developers to the direction of mistakes by printing warnings or errors. Also, automatic error detection is part of most modern Integrated Development Environments [1] (IDE) such as Visual Studio, Eclipse etc. thanks to static analysis.

There are, of course, some limitations to this technique, a more popular one relating to the Halting problem. As Alan Turing proved in 1936, it is not possible for an algorithm to determine if another will halt for all possible inputs. Anders Møller and Michael I. Schwartzbach best put it in their Static Program Analysis book as follows: [1, automated reasoning of software generally must involve approximation]. However, this technique can provide **guarantees** about specific execution paths in a program. The most important here is to not provide false positives, as that could result in optimizing something that should not be optimized or, even worse, change the semantics of a program.

The latter is an ethic similar to the Principle of Least Privilege in Program Security - by default, when doing static analysis, we assume that nothing can be optimized, i.e.

[1] This is also true for interpreted languages such as R or Python, although detection is not as comprehensive as it is for compiled, statically typed languages

worst case possible. As we perform analysis, we build guarantees over what can be optimized by ensuring that specific properties are respected, such as specific flows in the control flow graphs or specific variable states in data flow analysis.

Some straightforward examples of what this techniques seeks is:

- Are there declared variables that are not used?
- Are there assignments for variables that are not referenced anymore after the assignment?
- Are there mathematical operations executed that are not referenced and do not change the state of the program?
- Are there **unreachable flows** in the program execution, i.e. dead code?
- Are there `if` conditions that can be computed at compile time? If so, can they be directly replaced with true / false body?
- Are there duplicated expressions that can be computed only once? If so, can they be computed at compile time?

The above represent common optimization possibilities across programming languages and they represent a small subset in this domain. Depending on the properties of the programming language and the virtual machine that will run the generated bytecode, additional proprietary analysis may be done at compile time.

An example of the benefits of this analysis technique can be seen in two modern Machine Learning technologies, JAX and IREE. These perform computation on tensors, therefore widely used for differentiation operations in neural networks. Since in neural networks the data shape is often known, these technologies can trace the data shapes of the tensors throughout the computation pipeline. This helps with handling memory objects more efficiently, pre-allocating memory buffers, re-using them when necessary and apply parallel computation on memory blocks. This can happen both with Ahead Of Time Compilation (AOT) and Just In Time compilation (JIT), but it of course [8, requires array shapes to be static & known at compile time].

4.1 Control Flow Graphs

4.1.1 Overview

Control Flow Graphs (CFG) are esentially directed graphs $G = (V, E)$, where V is the set of nodes (**basic blocks**) and E determine the edges (**control flows**) between the nodes. They act as data structures used to analyse the execution flow in computer programming by following the edges between nodes. That is, it is not necessary for all of the edges to be visited during a program execution, and some nodes can never be visited, regardless of the program input. Since the whole program semantics are embedded into such a graph, we can safely affirm that it represents yet another intermediate representation of a computer program, hence why the operation of CFG generation is **symmetric** - we can reconstruct a computer program from a CFG.

CFGs are usually applied in situations of **flow-sensitive analysis** and where the order of the instructions matter.

4.1.2 Properties

We call a graph node v a **basic block**. The property of a node is that the code instructions it represents are executed sequentially, with no **interrupting flows** between these. This translates to no `JUMP` instructions in assembly code. We can easily state that any basic block v containing n instructions can be split into n basic blocks $v_1, v_2, ..., v_n$ since the property of no interrupting flows will be inherited by the "child nodes".

The first instruction in such a node is called a **leader**. Since the transition between nodes is made by jump instructions, the leaders translate to the first assembly instructions after jump instructions. Having observed this, we state that a basic block i starts at leader L_i and ends before leader L_{i+1}, property which can be used to generate the basic blocks of a CFG [11].

Considering a single CPU, the execution of a program means executing assembly operations **sequentially** on the same processor, hence why a CFG is a directed graph. The edge direction gives us the order of instruction execution, and cycles can be present

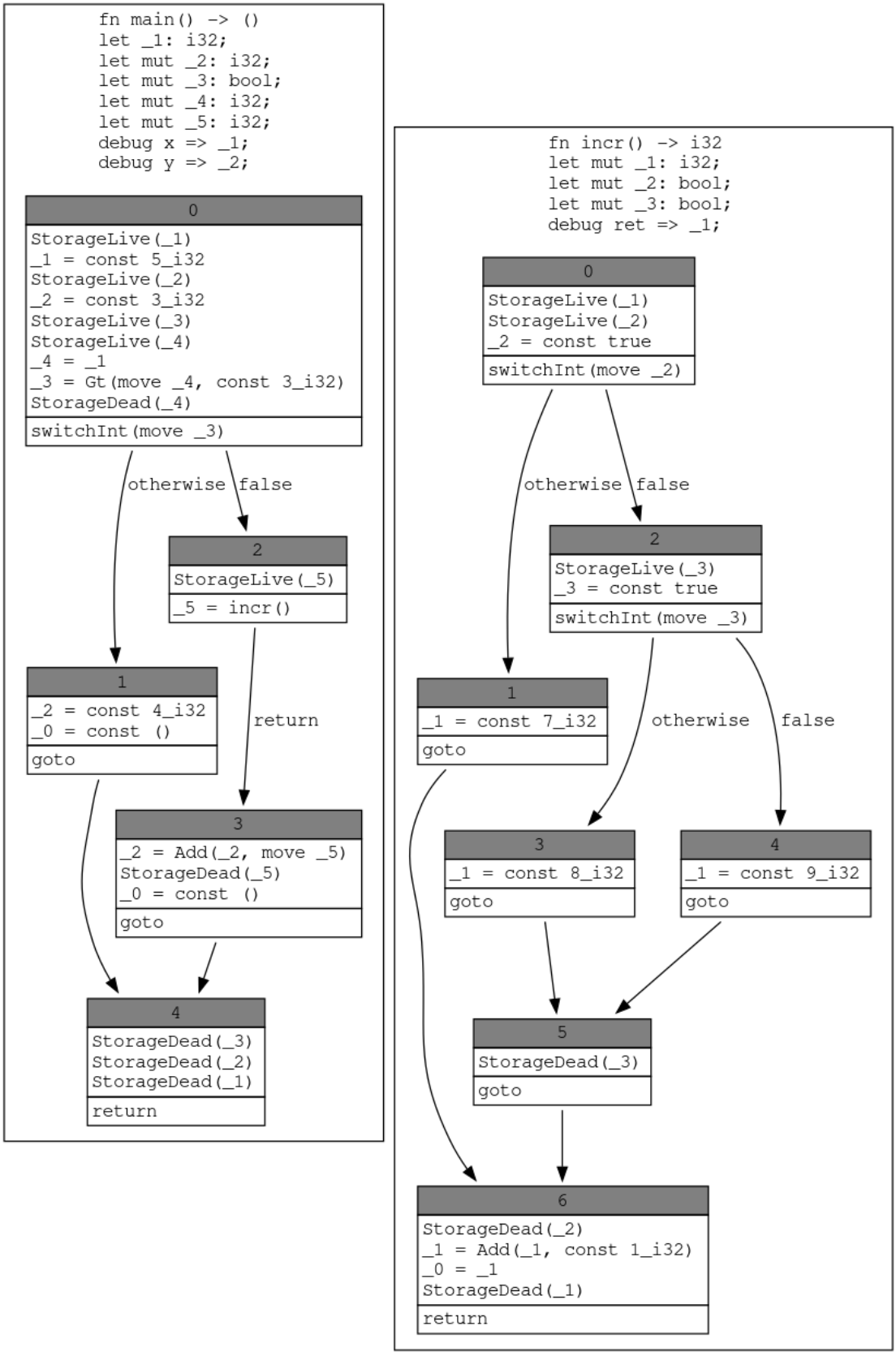

Figure 4.1: Example of a CFG built on top of a simple Rust program. The entry blocks allocate memory for variables, while the exit block act as garbage collection and terminate the execution flow. Source: Wikipedia

in this graph (eg. loops). Considering a graph node v, we have the following:

$deg^+(v)$, the inner degree (indegree) of the node, and

$deg^-(v)$, the outer degree (outdegree) of the node.

The inner degree $deg_+(v)$ correspond to all of the possible flows that precede the execution of v, so we have

$$deg^+(v) = |\{u, \exists e \in E, e = (u, v), u \in V\}|$$

We call v an **entry block** if $deg^+(v) = 0$, meaning there is no preceding execution flow for v, therefore that is the start of the program. Similarly, we call v and **exit block** if $deg^-(v) = 0$, where

$$deg^-(v) = |\{u, \exists e \in E, e = (v, u), u \in V\}|$$

Exit blocks introduce **termination flows** in the program[2].

4.1.3 Usage examples. Local vs. global optimizations

Code optimizations can be divided based on their context, local or global. Local optimizations are interested in inspecting small blocks of code, abstracting out the surrounding "environment" of the code, and these can further build their own representations of basic blocks. Since each instruction within a basic block works with one or more inputs, we can compute a DAG (Directed Acyclic Graph) that tracks input and intermediary variables usage throughout computation. The DAG can further be used to simplify the basic block, such as applying common subexpression elimination, by pruning the DAG whereever possible. The basic block is then rebuilt by walking the DAG and generating the code instructions according to the used dialect.

Another type of local optimization is Peephole optimization, which acts as a shifting window over small portions of a basic block. If basic blocks are granular enough, they can be applied on the whole block directly, however this depends on the CFG generation method. Peephole optimization can be used to apply common subexpression eliminator as well, however it is more concerned in finding specific patterns in the window of focus.

[2]It is possible to have termination flows in basic blocks that have a strictly positive outer degree due to non-jump instructions such as the `return` command.

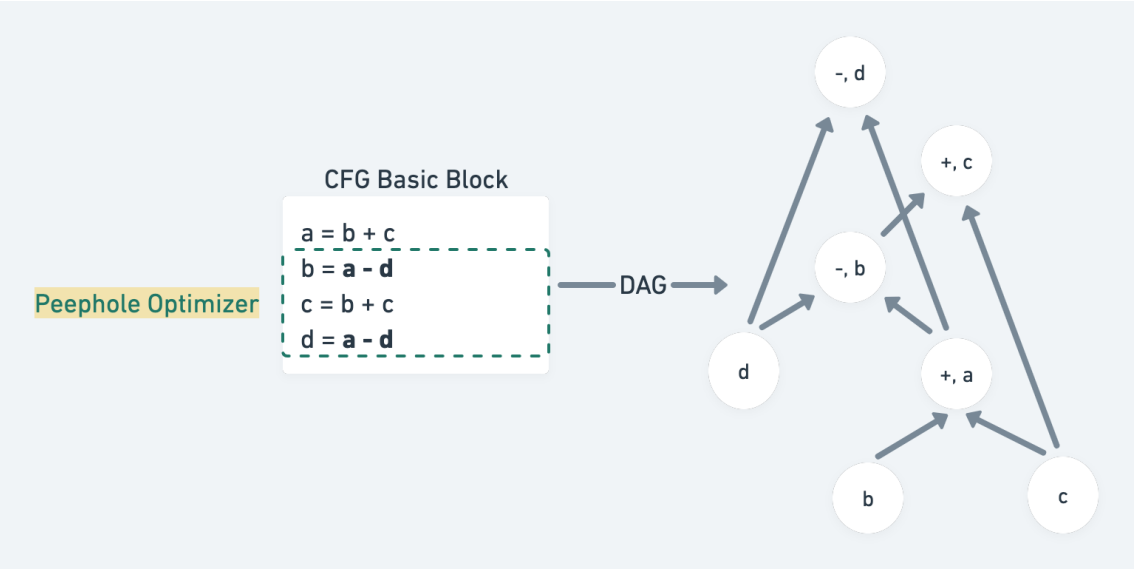

Figure 4.2: Representing the inputs of a basic block using DAGs. Example of the peephole optimizer and where common subexpression eliminator could be used.

Global optimizations [11, need to analyze the entire control flow graph of a program], such as figuring out if a computed expression, passed as a parameter through the execution flow is actually ever used. In order to judge such a case, we need to walk multiple paths in a CFG and identify whether there are any basic blocks using the given expression (parameter, variable etc.).

4.1.4 Limitations

Graph Completeness. Resolutions.

4.2 Data Flow Analysis

4.2.1 Overview

Data flow analysis is one of the dominant techniques of static analysis for reasoning about the flow of data in the program. This expands to the different kinds of data: variables, expressions and constants. It is impossible for a software analysis program to guarantee **soundness, completeness and termination** at the same time. The compromise that data flow analysis does is to sacrifice completeness, making it **incomplete**. That is, it will report all facts that could occur during program execution, but also "false positives", i.e. facts that will never occur in program runs.

Data flow analysis is not simply an AST walker.

TODO Correctness comes before performance!!!

The way completeness is sacrificed is by abstracting the control flow conditions.

Soundness informally means [12, that the flow functions map abstract information before each instruction to abstract information after that instruction in a way that matches the instruction's concrete semantics]. That is, let's say we have a problem which we represent by a pair of the input and output $P = (I, O)$. We define an abstractization function DF that maps the concrete input I to the abstract domain for the problem P:

$$DF(I) = I_a$$

and does the same for the output O:

$$DF(O) = O_a$$

In order to have a sound analysis, the data flow analysis must correctly perform modifications to the abstract input I_a, according to the correct machine semantics of the program P (eg. x86_64 ASM instructions) such that after each instruction, a transition function σ is applied, such that $S = I_a \rightarrow S = \sigma(S) \rightarrow \cdots \rightarrow S = \sigma(S)$. If the analysis is kept, then $S = O_a$.

TODO Soundness, completeness and termination.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.14;

contract DataFlowAnalysis {
    uint256 persistent_var;

    function variableTracing(uint256 n) public pure returns (uint256) {
        uint256 x; // x is 0
        if (n > 10) {
            x = 1; // x is 1
        } else {
            x = 2; // x is 2
        }
        // x is in {1, 2}
        return x;
```

```
16            }
17        }
```

Listing 4.1: Simple example of how data flow analysis is used to trace the values of a variable through the execution flow.

4.2.2 Examples

One example is null pointer safety in Dart, which has been very positively received by the developer community as a [9, boost to productivity]. Null pointers are often the cause of runtime exceptions, making an application crash without any sign in advance. What null pointer safety does is that it tells the compiler that a variable may or may not be null through the special "?" symbol: `String? hello = "world";`. This way, the compiler can perform data flow analysis and warn the developer in case a null pointer exception might be encountered. It is best put by the Dart documentation itself: [10, runtime null-dereference errors turn into edit-time analysis errors.]

Chapter 5

YUL Optimizer Deep Dive

5.1 Optimization flow

As seen in the solidity compilation flow 3.3, solidity optimizes its code via an **optimizer suite** which runs against the YUL IR. This consists of currently 32 documented steps [3], followed by a series of bytecode (opcode) optimizations which we will not focus on in this chapter. Some of the optimization steps are required to be run first, as they impose a specific structure to be further used and also make it safe to run arbitrary sequences of optimization steps [3], and are run in the following order:

1. **Disambiguator**: returns a copy of the YUL AST where each variable has a unique identifier. Therefore, an AST walker can maintain a global state with regards to all of the identifiers in the code, even if we have variables with the same name in different parts of the program. This property is maintained by any subsequent step and there are no identical identifier names introduced while optimizing.

2. **FunctionHoister**: moves all function definitions to the topmost block to enable isolated optimization of function definitions.

3. **FunctionGrouper**: reorders code, such that the YUL code is of the form $I\ F\ \ldots$, where I are instructions of the (possibly empty) entry block in the CFG and F contain **un-nested** function definitions.

4. **BlockFlattener** flattens nested blocks of code of form $\{\{B\}\}$ to $\{B\}$ if possible, i.e. if no interrupting control flows appear between the code blocks.

Optimization makes a big difference. In code sample 5.1 we compute gas cost to re-declare a variable inside a for loop. When not optimized, the execution gast cost estimates for running `declareVariableOutsideLoop` is 415250 gas and `declareVariableInsideLoop` is 420223 gas. This is expected, since the latter re-declares the same variable in every loop iteration.

When optimized, however, gas usage is almost four times better, with `declareVariableOutsideLoop` estimated to 119194 gas and `declareVariableInsideLoop` to 119211. An important note here is that the gas difference comes from the **function dispatch**, as described by Hari Mulackal [13], i.e. the order of the functions in a contract introduce a small overhead. Specifically for this case, the optimization comes from the LoopInvariantCodeMotion step, which moves declarations (not assignments) outside the for loop.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.14;
3
4   contract VariableLoop {
5       function declareVariableInsideLoop() public pure {
6           for (uint i = 1; i <= 1000; i++) {
7               uint x = i * i;
8           }
9       }
10
11      function declareVariableOutsideLoop() public pure {
12          uint x;
13          for (uint i = 1; i <= 1000; i++) {
14              x = i * i;
15          }
16      }
17  }
```

Listing 5.1: Code example of bad variable declaration inside a for loop

Order matters when running the optimization steps. In version 0.8.14, the default order [1] of the suite is

[1]Optimization order is given by the step abbreviations, where each letter corresponds to an optimization phase. The mapping is described in the Optimization Step Sequence section in the documentation.

```
    dhfoDgvulfnTUtnIf[xa[r]EscLMcCTUtTOntnfDIulLculVcul [j]Tpeulxa[rul]xa[r
]cLgvifCTUca[r]LSsTFOtfDnca[r]Iulc]jmul[jul] VcTOcul jmul
```

In the above order, the optimizers between brackets [3, will be applied multiple times in a loop until the Yul code remains unchanged or until the maximum number of rounds (currently 12)]. We can immediately see that some of the optimizers are run more than once, the reasoning being that each optimizer may change the YUL IR, therefore giving new optimization opportunities to other optimizer. This is the case for UnusedAssignEliminator (r abbreviation) and UnusedPruner (u abbreviation). For example, let's optimize the following code in 2 rounds, first applying UnusedPruner and then UnusedAssignEliminator, then applying UnusedAssignEliminator and then UnusedPruner [2]:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.14;

contract Termination {
    uint persistent_var;
    event FunctionCalled(string f);

    function assignBeforeTermination() public {
        emit FunctionCalled("assignBeforeTermination");
        uint balance;
        balance = 7919;
        return;
    }
}
```

Listing 5.2: Example of a smart contract with a redundant assignment and variable declaration

In the above figures 5.2 5.1, we have the same unoptimized YUL IR in the left-hand side, with the optimized YUL code on the right. Upon applying UnusedPruner, this step will look to remove any variable declarations that are not used. When applied first, it correctly identifies `let expr_1 := _4` as a redundant variable declaration and

[2]The corresponding commands are `solc --optimize --ir-optimized --via-ir --yul-optimizations 'ur' termination.sol` and `solc --optimize --ir-optimized --via-ir --yul-optimizations 'ru' termination.sol`

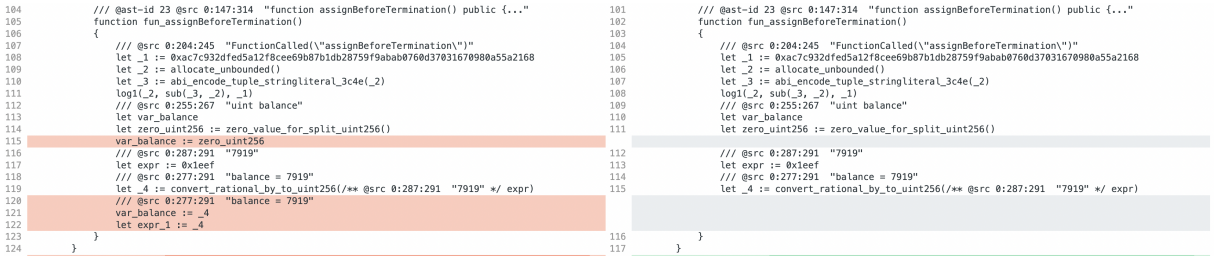

Figure 5.1: Unoptimized vs. optimized YUL IR for Solidity code 5.1, order of optimizers: UnusedPruner then UnusedAssignEliminator

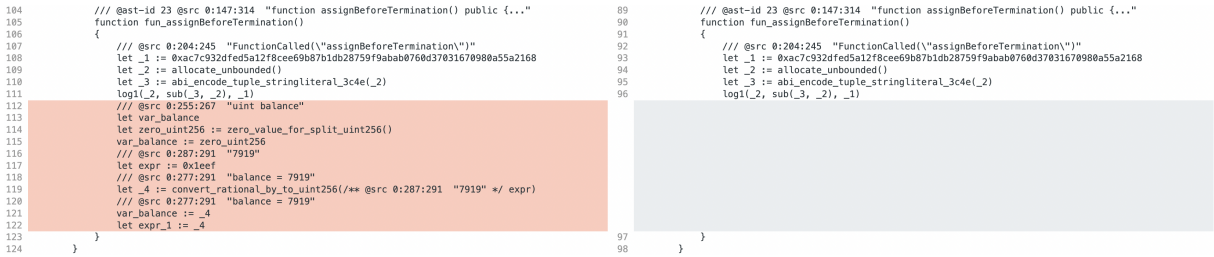

Figure 5.2: Unoptimized vs. optimized YUL IR for Solidity code 5.1, order of optimizers: UnusedAssignEliminator then UnusedPruner

it removes it. Afterwards, UnusedAssignEliminator runs and it finds `var_balance := zero_uint256` and `var_balance := _4` as redundant assignments and removes those as well. However, we can still notice some left-over assignments and variable declarations.

If, however, UnusedAssignEliminator is run first, it removes the same assignments (`var_balance := zero_uint256` and `var_balance := _4`), but then gives more room for UnusedPruner to act. Therefore, all of the variable declarations get removed, since their values are not used and they are redundant.

While this is a proof-of-concept example, optimizer steps frequently generate intermediary data objects (such as the SSATransform) to make local optimizations safe and possible (remember that corectness comes before optimization). Hence why these two optimizers are simple, yet effective.

5.2 UnusedPruner. UnusedAssignEliminator.

A very effective way of reducing gas usage is by removing any kind of redundant variable assignments or variable declarations, since it implies executing less instruc-

tions. This can be applied as both a local optimization and as a global optimization. In case we locally prune variable declarations, we must ensure that other control flows do not have references to the the left-hand side of the assignment. For example, considering 5.2, we can see that the assignment at line 3 can be removed, since `a` is immediately re-assigned. However, even though `a` gets re-assigned at line 10, we cannot remove assignment at line 4, because there is another control flow that uses the latter assignment, in the `if` statement.

```
1  {
2      let a
3      a := 1
4      a := 2
5      b := 2
6      if calldataload(0)
7      {
8          b := mload(a)
9      }
10     a := b
11 }
```

Listing 5.3: YUL IR code sample for variable assignments. Both variables a and b are used in a different control flow.

5.2.1 Variable states in UnusedAssignEliminator

UnusedAssignEliminator acts in two phases: one to collect information and one to prune redundant assignments. While doing data flow analysis, it assigns different states to the identifiers [3]: **undecided, used and unused**. It does so by acting as an YUL AST walker and visiting each VariableAssignment node. On the second pass, any assignment expression that is in the "undecided" state will be marked as "unused" and will be pruned from the AST, therefore from the final YUL IR code.

An assignment `a := b` will be marked as "used" only if there exists another expression E within the same basic block B or within a basic block B' such that we have a path in the CFG from B to B', where E contains `a` on the right-hand side. Similarly, if such an expression E does not exist, then the assignment can be marked as "unused".

[3]This optimizer only treats assignment where the left-hand side is a single identifier (variable)

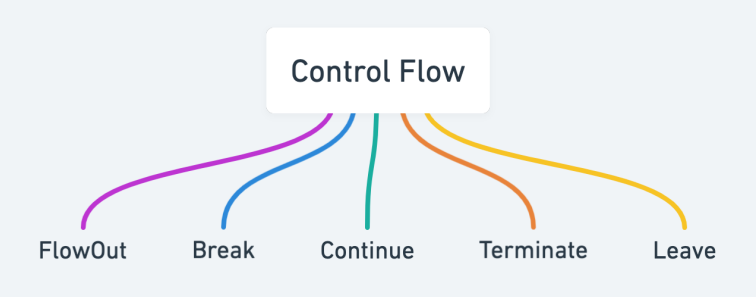

Figure 5.3: Control flow types in YUL IR

5.3 Handling termination flows

While getting acquainted with the compiler, several experiments have pointed out that the compiler does not treat all of the edge cases with regards to redundant assignments, especially when **termination flows** appear within basic blocks. YUL IR uses the EVM dialect where termination flows are given by the `return`, `revert`, `selfdestruct` and `invalid` instructions [4]. To understand the examples, we must note the different control flows that are present in the CFG for the YUL AST: FlowOut, Break, Continue, Terminate and Leave.

All of the above except FlowOut are applied to simple, one-line statements. FlowOut is the most complex one and it embeds the normal execution of the program, as well as the terminal execution of a program. It is applied "by default" to `for` loops [5], `if` statements, `switch` statements and functions that can revert, stop or exit normally upon calling them.

A successful attempt to "fool" the compiler was done by introducing a Terminate control flow after a redundant assign eliminator. The code sample is as follows:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.14;
3
4  contract Termination {
5      uint persistent_var;
6      event FunctionCalled(string f);
7      event BalanceIsEmpty();
```

[4] It suffices to focus on the YUL IR instructions, since the optimization flow does not work with Solidity code directly.

[5] In YUL IR, `while` loops get converted to `for` loops

```
8
9           function assignBeforeTermination(uint x) public {
10              emit FunctionCalled("assignBeforeTermination");
11              uint balance;
12
13              if (f1(x) == true) {
14                  balance = 10007;
15                  return;
16              }
17
18              if (balance == 0) {
19                  emit BalanceIsEmpty();
20              }
21              return;
22          }
23
24          function f1(uint x) public pure returns (bool) {
25              return x > 10;
26          }
27      }
```

Listing 5.4: YUL IR code sample for variable assignments. Both variables a and b are used in a different control flow.

The introduced termination flow is at line 15, right after the assignment to variable `balance`, which is then referenced inside the second `if` condition. The usage of `f1` functions is necessary to make it impossible for the compiler to compute the `if` condition's value at compile time, i.e. introduce a **non-deterministic** flow. The compiler seens `balance` as referenced in a different control flow, therefore it will not remove assignment at line 14. We can easily see, however, that if the condition `f1(x) == true` does not stand, then the second if will always evaluate to true, hence why the StructuralSimplifier optimizer could be used to simplify things further and entirely remove the second if condition, since it will always evaluate be true if the termination flow is not executed.

To handle this edge case, we can simplify the problem of determining wether a basic block contains a termination flow or not. Figure 5.5 shows how a `leave` statement could be following by valid YUL IR statements (i.e. dead Solidity code after a `return` instruction). We use the DeadCodeEliminator step for this case to ensure that there are

```
176                 /// @src 0:320:399  "if (f1(x) == true) {..."
177                 if expr_3
178                 {
179                     /// @src 0:363:368  "10007"
180                     let expr_4 := 0x2717
181                     /// @src 0:353:368  "balance = 10007"
182                     let _5 := convert_t_rational_by_to_t_uint256(/** @src 0:363:36
    8  "10007" */ expr_4)
183                     /// @src 0:353:368  "balance = 10007"
184                     var_balance := _5
185                     let expr_5 := _5
186                     /// @src 0:382:389  "return;"
187                     leave
188                 }
```

```
171                 /// @src 0:320:399  "if (f1(x) == true) {..."
172                 if expr_3
173                 {
174                     /// @src 0:363:368  "10007"
175                     let expr_4 := 0x2717
176                     /// @src 0:353:368  "balance = 10007"
177                     let _5 := convert_t_rational_by_to_t_uint256(/** @src 0:363:36
    8  "10007" */ expr_4)
178                     /// @src 0:353:368  "balance = 10007"
179                     var_balance := _5

180                     /// @src 0:382:389  "return;"
181                     leave
182                 }
```

Figure 5.4: Optimized YUL IR using UnusedAssignEliminator then UnusedPruner. UnusedAssignEliminator does not handle termination flows while pruning

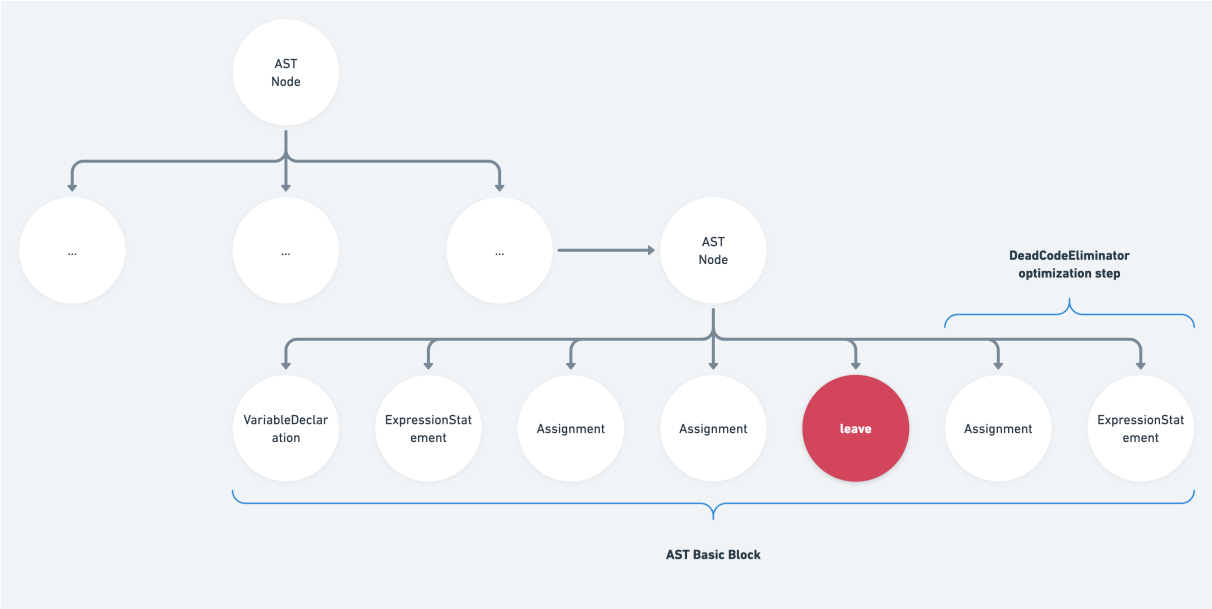

Figure 5.5: Example of termination flow inside a YUL AST's basic block node. DeadCodeEliminator is used to prune unreachable code

no instructions following an instruction that terminates the execution.

With this simplification in mind, the edge case can be handled by enhancing the `UnusedAssignEliminator.visit` function for Basic Blocks in the YUL AST. That is, when visiting the block, we make a copy of the current recorded assignments (outside of the current basic block) such that after visiting we can compute the new assignments - this includes variable declarations. If basic block B's body trailing statement induces a termination flow, then we can safely remove all of the new assignments made in that block, even if they are referenced in another block of the CFG.

Some special cases must be considered however, such as **state variables** which are persistent within the contract. These are defined in Solidity as members of the smart contract. Obviously, any assignment to such a persistent variable p **should not** be

Figure 5.6: Full optimizer suite ran against code sample 5.3. The right hand side YUL IR handles termination flows within basic blocks for variable assignments and declarations.

removed, regardless of the control flows that follows it. The only case in which we can safely remove them is if they consist unreachable code, which the DeadCodeEliminator will handle. We therefore restrict the behaviour of the above enhancement to only local basic block variables or to the inputs of the basic block [6].

The Solidity compiler reports that the estimated gas usage for the contract construction for code sample 5.3 went down from 67723 to 65723, a saving of 2000 gas. Also, the execution of the `assignBeforeTermination` function went down from 2585 to 2565, which is 20 gas less. The result of the enhancement can be seen in figure 5.6 where the full optimizer suite was ran against the same code sample, with and without the above enhancement. This also enabled StructuralSimplifier to simplify code further, completely removing the second `if` condition and un-nesting the emitting of the `BalanceIsEmpty` event.

5.4 Pruning redundant termination flows

As stated in the previous chapter, running an optimization step X may give more opportunities to optimization step Y to further enhance the YUL IR (therefore the re-

[6] This is actually easy to do because persistent variables are kept in memory. Their values are updated via function calls, which are not recognized by the AST walker as variable assignments / declarations.

sulted bytecode). This was the case for the above enhancement, where experiments pointed out that if a basic block contains a trailing `if` statement, then pruning code inside the `if`'s body may result in a single trailing, redundant `leave` instruction within the body. The expected basic block pattern for this case is

```
{ ... if (C) { leave } }
```

after the removal of redundant code has been done.

A good solution for this edge case was changing how the StructuralSimplifier works and make it look for `leave` statements within an `if`'s body. If that's the case, then the `leave` statement is simply removed. This enabled for cascading enhancements, and the StructuralSimplifier will further completely remove the `if` condition and body and replace it with the operations inside the condition itself. The removal of the `leave` statement did not improve gas usage significantly, but the cascading optimization aspect is what enables true optimization opportunities here.

Chapter 6

Validation and benchmarking

In the optimization world, corectness comes before performance. Ensuring correctness of compiler enhancements is mandatory before even discussing about gas usage improvements. In order to ensure that the modifications done to the compiler do not alter the semantics of a smart contract, validation was done using unit testing, contract fuzzing and tests against the Etherscan dataset.

The advantage of forking the Solidity eco-system and integrating the enhancements directly in the official codebase is that a first validation pass can be done by simply running the existing unit tests, and then extending those. All of the unit tests have passed. An additional validation was done by fuzzing, which in software development acts as a tool that generates random data for automated testing purposes. In the case of Solidity, fuzzing is used to construct valid, random smart contracts and make sure that they still compile. This also helps with regression testing, as fuzzing may ocassionally construct valid smart contracts that cannot be compiled anymore.

A testing experiment was done using the online Etherscan dataset, which contains a series of verified smart contracts. The experiment meant acquiring all of the addresses for the 5000 [1] available verified constracts from the Etherscan dataset, then downloading the source code for all of the contracts. Then, all of the smart contracts using lower versions of the Solidity compiler were discarded, i.e. files that did not match the pragma of `pragma solidity ^0.8.14` in our case. As a validation step, all of the contracts that were compilable with the official 0.8.14 compiler release also needed to be compilable with the enhanced 0.8.14 compiler, step which passed.

[1] 5000 is the imposed limit by the Etherscan API

While validation was successful, contracts from the Etherscan network (at the time of writing this thesis) showed no benefits in the gas estimates of the contract construction. This might also be because most of the contracts returned "infinite" as a gas estimate, which is due to instructions such as loops in the code or recursive function calls. This is to be expected, since the edge cases described in this thesis are most likely the result of code mistakes on the developers' side, mistakes which often get caught during steps of code review. However, it is not excluded that future changes to the optimizer codebase will take advantage of the enhancements described in the previous chapter.

Conclusions

Solidity is still a place where "construction is in progress". What's common across optimizing all of the versions of this programming language is common software techniques, known as static analysis. Control Flow Graphs are a center piece of this mechanism and represents a good starting point, even more valuable when augmented by ASTs and other intermediate representations of the code, such as op code or YUL.

State of the art in Smart Contract Optimization is still receiving inputs, modifications, enhancements from many parts as parties, as the programming language itself is still under heavy development. https://www.overleaf.com/project/627ab6f7ebec3330916a293b

Bibliography

[1] Static Program Analysis, Anders Møller and Michael I. Schwartzbach, 2022

[2] Solidity Language, Solidity Team, 2022

[3] The Optimizer, Solidity v0.8.14 Official Documentation, Solidity Team, 2022

[4] Ethereum Smart Contracts up 75% to Almost 2M in March, Coin Telegraph, 2020

[5] Solidity Instrumentation Framework (SIF), Chao Peng, Sefa Akca, Ajitha Rajan, University of Edinburgh, 2019

[6] Slither: A Static Analysis Framework For Smart Contracts, Josselin Feist, Gustavo Grieco, Alex Groce, 2019

[7] YUL Motivation and High-level Description, Solidity Team, 2022

[8] How to think in JAX, The JAX authors, 2020

[9] Sound null safety in Flutter 2: Why is it a major milestone?, Proxify, 2021

[10] Sound null safety, Dart Dev, 2020

[11] Control Flow Graphs, Program Analysis and Optimization course (DCC888), Fernando Magno Quintão Pereira, Departamento de Ciência da Computação, 2022

[12] Claire Le Goues, 15-819O: Program Analysis (Spring 2016) course, CMU, 2016

[13] Solidity Optimizer, Solidity Summit, Hari Mulackal, 2022