Verificación y Certificación de Optimizaciones de EVM
Verification and Certification of EVM Optimizations

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Fernando Isaías Leal Sánchez

Directores
Samir Genaim
Enrique Martín Martín

Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Verificación y Certificación de Optimizaciones de EVM
Verification and Certification of EVM Optimizations

Memoria del trabajo de fin de grado de ingeniería informática del Doble Grado de Ingeniería Informática y Matemáticas

Autor

Fernando Isaías Leal Sánchez

Directores

Samir Genaim

Enrique Martín Martín

Convocatoria: Mayo 2024

Grado en Ingeniería Informática y Matemáticas

Facultad de Informática

Universidad Complutense de Madrid

23 de mayo de 2024

Dedicatoria

A mis padres y hermano por siempre estar ahí cuando los necesito. El compromiso es recíproco. A mi abuelita Maritere, por creer en mi y ser mi crítica más fiable. El cariño es mutuo. Finalmente, a todos mis amigos, por demostrarme constantemente que hay algo más en esta vida además de lo académico.

Acknowledgements

To the authors of Logical Foundations [9] for the magnificent resource they have made freely available to the world. I owe them all my knowledge of Coq. To Johannes Hostert, Ike Mulder, Théo Winterhalter, Meven Lennon-Bertrand, Pierre-Marie Pédrot, Pierre Roux and Kenji Maillard, for their time and help when I asked questions in Coq's Zulip chat. And finally, to my thesis directors Samir Genaim and Enrique Martin-Martin for helping me through the process of writing my first academic paper.

Resumen

Verificación y Certificación de Optimizaciones de EVM

La optimización del *bytecode* de la máquina virtual de Ethereum (EVM) presenta retos y oportunidades únicos debido a sus requisitos específicos, que incluyen consideraciones como el coste de ejecución y el tamaño del ejecutable compilado. A pesar de sus potenciales beneficios, la optimización en este ámbito sigue siendo limitada, principalmente debido a los altos riesgos asociados con los programas EVM, llamados contratos inteligentes, donde incluso errores menores pueden resultar en pérdidas financieras significativas. Para abordar estas preocupaciones, el proyecto FORVES desarrolla un verificador, aprovechando las técnicas de verificación formal para garantizar que el código de bytes optimizado conserve su semántica original. Al emplear Coq, un asistente de pruebas y lenguaje de programación rigurosamente verificado, FORVES aumenta la confianza en los procesos de optimización, trasladando la carga de la prueba del optimizador al verificador. Este trabajo ocurre en el entorno de `FORVES2`, un sucesor del proyecto anterior que incorpora información contextual para evaluar la equivalencia del código. En concreto, el proyecto pretende diseñar un verificador de implicaciones capaz de comprobar si ciertas restricciones octogonales se cumple en contextos específicos, empleando para su validación un algoritmo de cálculo del cierre transitivo.

Palabras clave

Coq, Restricciones octogonales, Programación funcional, Smart contracts, Ethereum, EVM, Blockchain

Abstract

Verification and Certification of EVM Optimizations

The optimization of Ethereum Virtual Machine (EVM) bytecode presents unique challenges and opportunities due to its distinct requirements, including considerations such as execution cost and compiled binary size. Despite its potential benefits, optimization in this realm remains limited, primarily due to the high stakes associated with EVM programs, or smart contracts, where even minor errors can result in significant financial losses. To address these concerns, the `FORVES` project introduces a verifier, leveraging formal verification techniques to ensure optimized bytecode retains its original semantics. By employing Coq, a rigorously verified proof assistant and programming language, `FORVES` enhances trust in optimization processes, shifting the burden of proof from the optimizer to the verifier. This project outlines the development of `FORVES2`, an advanced iteration, which incorporates contextual information to assess code equivalence. Specifically, the project aims to devise an implication checker capable of verifying whether certain octagonal constraints hold in specific contexts, by employing a tightened transitive closure algorithm for validation.

Keywords

Coq, Octagonal restrictions, Functional programming, Smart contracts, Ethereum, EVM, Blockchain

Contents

1. Introduction **1**
1.1. Motivation 1
1.2. Objectives 3
1.3. Work plan 3

Introducción **7**
1.4. Motivación 7
1.5. Objetivos 9
1.6. Plan de trabajo 9

2. State of the art **13**
2.1. Blockchain 13
2.1.1. The Ethereum Virtual Machine 13
2.1.2. Optimization of smart contracts 16
2.2. Octagonal constraints 17
2.3. The Coq Proof Assistant 18
2.3.1. Coq as a functional programming language 18
2.3.2. Coq as a theorem prover 23
2.3.3. The Curry-Howard correspondence 29

3. Modelization of the problem in Coq **33**
3.1. Semantic equivalence of programs 33
3.2. Representation of constraints 35
3.3. Implication between constraints 36
3.4. Implication checker 37

4. Implication checker from transitive closure **41**
4.1. Tightened transitive closure of octagonal constraints 41
4.2. Implementation of the implication checker 44

5. Conclusions and Future Work **51**
5.1. Future work 53

Conclusiones y Trabajo Futuro **55**
5.2. Trabajo futuro 57

Bibliography **59**

Chapter 1

Introduction

"Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some current state. It is this current state which we accept as the canonical "version" of the world of Ethereum."

— Ethereum Yellow Paper [13]

1.1. Motivation

In the field of code optimization, Ethereum Virtual Machine (EVM) bytecode stands as a language with much to benefit from, as it has some unique requirements which open up new dimensions for optimization. For general purpose programming languages the optimization's focus are usually on execution time or memory usage. However, in the case of programs compiled to EVM bytecode, called *smart contracts*, one must also take into consideration other factors, such as the size of the compiled binary or the cost of the programs execution. Every EVM instruction requires a fee (called *gas*[1]) to be paid for its execution, and these prices can vary greatly in magnitude.

However, the optimization of EVM bytecode is not as widespread as one could imagine. This is in part due to the higher stakes that EVM programs tend to work under. A bug in a *smart contract* usually has a greater impact than most other programs, often resulting in large monetary loses[2]. It is therefore understandable that *smart contract* developers do not want to risk being responsible of the semantics of their code being contaminated by a third party's tool.

To mitigate this risk, the `FORVES` [3] project aims to develop a verifier which is able to guarantee that an optimization of a jump-free sequence of EVM instructions (that is, without pieces of code that are conditionally executed) retains the same semantics as the original unoptimized version. `FORVES` works by computing the resulting state after the execution of both sequences and simplifying them to prove

[1] See `https://www.evm.codes/`

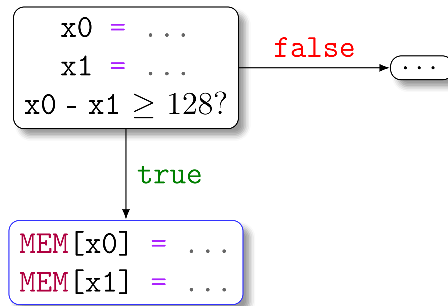
[2] See `https://www.gemini.com/cryptopedia/the-dao-hack-makerdao`

[3] FORmaly Verified EVM optimizationS `https://github.com/costa-group/forves2/`

their equivalence. This, however, does not by itself suffice to appease the worries of the *smart contract* developer, since we have merely pushed the burden of trust from the optimizer to the verifier. If the verifier were to have a bug, it could guarantee that an erroneous optimization is in fact valid, impacting its users.

The solution is to, again, shift the burden of proof to another tool. For the `FORVES` project the verifier has been written in Coq, a proof assistant and programming language which allows its users to define a formal specification and prove that their program adheres to it. The benefit of using a tool such as Coq is that the burden of proof is placed on a small core subset of its features, which have been deeply scrutinized by mathematicians, and out of which all the other functionalities are built. Therefore, to trust that a Coq-certified program is correct is equivalent to trusting that the Coq kernel is correct, which is a much greater assurance[4]. Many different projects have taken advantage of this fact to develop pieces of critical software which can be trusted upon [3, 6, 7, 1, 4]. A brief description of these projects can be found in Chapter 2.

The purpose of this project is to aid in the development of `FORVES2`, a successor of `FORVES` which improves on its reasoning power by allowing it to take into account contextual information to decide if two pieces of code are equivalent. The jump-free sequences of instructions which `FORVES` takes as input are usually extracted from a complete program, which offers additional information about how it is used and what kind of values it receives. In particular, to apply some optimizations it is necessary to be able to derive certain constraints from this contextual information. Consider the following example.



The blue block is only reachable if the jump condition is met, which means that `x0` $\geq$ `x1` $+128$. This information can be useful when optimizing a piece of code, and must therefore be taken into consideration when certifying that two sequences of instructions are semantically equivalent. For example, knowing that `x0` $\geq$ `x1` $+128$ could imply that the addresses of memory pointed to by `x0` and `x1` are disjoint, and therefore that the memory accesses can be reordered without altering the final result.

The contextual information which `FORVES2` is interested in are constraints of the form $x + d \leq y + d'$ where x and y represent variables which point to EVM values (which themselves are a subset of the integers) and d and d' are integer constants. In particular, from the previous constraints we can also derive the constraints of the form $x \geq y$, $x = d$ or $x < y + d'$ by taking advantage of the symmetry of the order

[4] There are even efforts to verify Coq in Coq! [11]

relation, the combination of multiple constraints or the fact that $x \leq k$ for x and k integers implies that $x < k + 1$.

Therefore, this project aims to develop a constraint solver which is able to prove that a given constraint holds for a given context. To do so, we transform the constraints into a special type, called octagonal constraints, and we employ the tightened transitive closure algorithm to finally check if a constraint can be derived from it.

1.2. Objectives

As previously stated, the main objective of this project is developing a constraint solver which would allow the usage of contextual information to enable specific simplifications in the `FORVES2` project that require some conditions to hold true to be applied. Furthermore, in spirit with the rest of the project, this constraint solver must be verified in Coq. To do so, we set the following sub-objectives:

1. Study Coq as a tool for the development of verified software.
2. Model the problem in Coq.
3. Implement an implication checker which is able to determine whether a constraint can be derived from a set of contextual information.
4. Verify the implication checker in Coq.

1.3. Work plan

Each objective was further subdivided in various tasks. These tasks were then distributed between the allotted time frame for the project.

1. Study Coq as a tool for the development of verified software.

   1.1. Read Logical Foundations [9] up to Chapter 6 - Logic.

   1.2. Complete Logical Foundations [9].

2. Model the problem in Coq.

   2.1. Define the structure of the desired constraints.

   2.2. Define the correct notion of implication between constraints.

   2.3. Define what is an implementation checker.

3. Implement an implication checker which is able to determine whether a constraint can be derived from a set of contextual information.

   3.1. Implement a basic version of an implication checker, to ensure the model works as desired.

3.2. Implement a more powerful implication checker algorithm through a constraint solver.

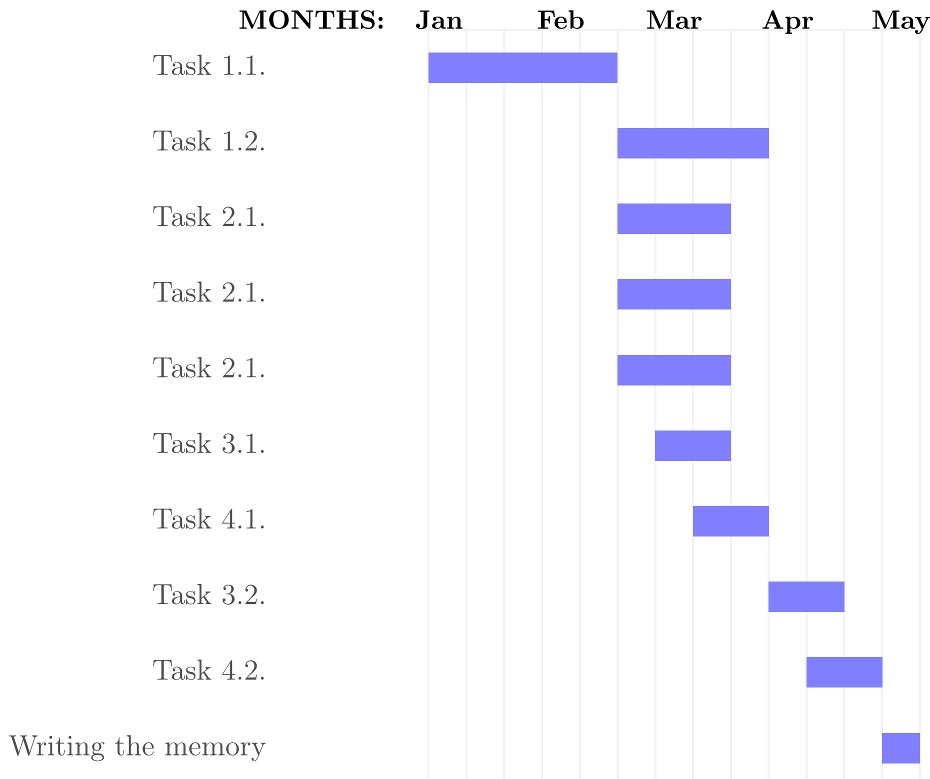
4. Verify the implication checker in Coq.

   4.1. Develop the theory to verify the basic implication checker.

   4.2. Develop the theory to verify the final implication checker.

The development of the project was planned to take place between January and May of 2024. The first two months were reserved to get familiar with the tools to be used, such as the Coq proof assistant and `FORVES2`. The following month was dedicated to model the problem in Coq, by deciding and stabilizing the definitions that were going to be used and implementing a basic implication checker to test how these definitions would fit in the bigger scope of the project. During this month various algorithms were also considered to implement the constraint solver. The implementation of the final implication checker was left for May, by developing the chosen algorithm in Coq and deriving the theory necessary to verify it. This theory ended up being developed independently of the definitions of the project, and we considered linking this theory with the rest of the project as an additional subtask, but we quickly realized this effort was out of scope for the project. See Section 5.1 for more information.

The final development cycle can be visualized in the following Gantt diagram.



The project is developed in `FORVES2`'s GitHub repository[5]. The files relevant to

[5] `https://github.com/costa-group/forves2`

this project are `constraints.v` and `octagon.v`, with the former using some terms defined in other files in the project.

Introducción

"Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some current state. It is this current state which we accept as the canonical "version" of the world of Ethereum."

— Ethereum Yellow Paper [13]

1.4. Motivación

En el campo de optimización de código, el *bytecode* de la máquina virtual de Ethereum (EVM) destaca como un lenguaje con mucho de lo que beneficiarse de la disciplina, debido a requisitos particulares del lenguaje que abren nuevas posibilidades para su optimización. En lenguajes de programación de propósito general el objetivo de las optimizaciones suele ser el tiempo de ejecución o el uso de memoria. Sin embargo, los programas compilados a *bytecode* de la EVM, conocidos como contratos inteligentes, uno debe tener en consideración nuevos factores como el tamaño del ejecutable compilado o el coste monetario de su ejecución. Toda instrucción de la EVM tiene un precio, conocido como *gas*[6], que debe ser pagado para ejecutarla, y estos precios pueden variar en gran medida según la instrucción.

Aún así, la optimización de *bytecode* de la EVM no está tan extendido como uno imaginaría inicialmente. Esto se debe en parte a los grandes riesgos bajo los que los programas de la EVM suelen trabajar. Un error en un contrato inteligente suele tener un mayor impacto que en otros programas, a menudo resultando en grandes perdidas monetarias[7]. Por lo tanto es comprensible que los desarrolladores de contratos inteligentes no quieran responsabilizarse de los posibles cambios en la semántica de su programa que una herramienta de terceros pueda introducir.

Para mitigar el riesgo, el proyecto `FORVES`[8] pretende desarrollar un verificador capaz de garantizar que una optimización de una secuencia de instrucciones de la EVM sin saltos conserva el mismo significado que su versión sin optimizar. `FORVES` consigue esto calculando el estado resultante de la ejecución de ambas secuencias y simplificándolas para probar su equivalencia. Sin embargo, esto no es suficiente para

[6] Ver `https://www.evm.codes/`

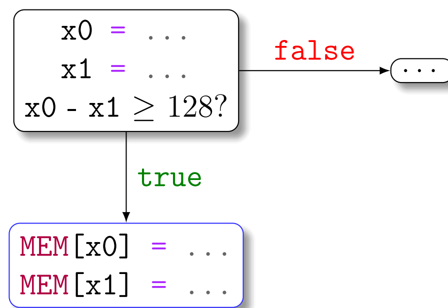
[7] Ver `https://www.gemini.com/cryptopedia/the-dao-hack-makerdao`

[8] FORmaly Verified EVM optimizationS `https://github.com/costa-group/forves2/`

aplacar las preocupaciones de los desarrolladores de contratos inteligentes, puesto que simplemente hemos trasladado la carga de la prueba del optimizador al verificador. Es decir, si el verificador tuviera un error, este podría garantizar que una optimización errónea es válida.

La solución es volver a trasladar la carga de la prueba a otra herramienta en la que sí podamos confiar. Para el proyecto `FORVES` el verificador se ha escrito en Coq, un asistente de pruebas y lenguaje de programación que permite a sus usuarios definir una especificación formal y probar que su programa la cumple. Los beneficios de usar esta herramienta es que la carga de la prueba se concentra en un subconjunto de funcionalidades que han sido cuidadosamente estudiadas por matemáticos y sobre las que el resto de sus funcionalidades se basan para garantizar su correctitud. Es decir, para confiar en que un programa certificado por Coq es correcto solo hace falta confiar que el núcleo de Coq es correcto, el cual es mucho más fiable[9]. Muchos otros proyectos se han aprovechado de este hecho para desarrollar programas críticos en los que se pueda confiar[**?**]. Una breve descripción de estos proyectos, en inglés, se puede encontrar en el Capítulo 2.

El objetivo de este proyecto es ayudar en el desarrollo de `FORVES2`, un sucesor de `FORVES` que mejora sus capacidades de razonamiento permitiéndole tener en cuenta la información del contexto para determinar si dos secuencias de instrucciones son equivalentes. Las secuencias de instrucciones libres de saltos de la EVM que `FORVES` recibe vienen normalmente de un programa más grande, el cual cuenta con información adicional sobre como se usa esta secuencia y que tipo de valores recibe. En particular, para realizar ciertas optimizaciones es necesario poder deducir ciertas restricciones de la información del contexto. Veamos un ejemplo.



El bloque azul solo se puede alcanzar si la condición de salto se cumple, lo cual a su vez nos indica que `x0`$\geq$`x1`$+128$. Esta información puede ser útil cuando se optimiza una parte del código, y por lo tanto se debe tener en consideración si queremos certificar que dos secuencias de instrucciones son semánticamente equivalentes. Por ejemplo, sabiendo que `x0`$\geq$`x1`$+128$ es cierto podemos deducir que las regiones de memoria a las que apuntan `x0` y `x1` son disjuntas, y por lo tanto los accesos a memoria se pueden reordenar sin alterar el resultado final.

La información del contexto de interés para `FORVES2` son restricciones de la forma $x + d \leq y + d'$ donde x e y representan variables que apuntan a valores de la EVM, que a su vez son un subconjunto de números enteros, y d y d'' son constantes enteras. Usando restricciones como la anterior también se pueden derivar otras restricciones

[9]Hay incluso un proyecto para verificar Coq en Coq. [11]

como $x \geq y$, $x = d$ o $x < y + d'$ usando la simetría de la relación de orden, la combinación de múltiples restricciones o el hecho de que si $x \leq k$ con x y k enteros entonces $x < k + 1$.

Por lo tanto, el objetivo de este trabajo es desarrollar un resolutor de restricciones que sea capaz de probar que una restricción dada se cumple para un contexto específico. Para ello transformamos las restricciones a un tipo específico, llamadas restricciones octogonales, y usamos el algoritmo de la clausura transitiva para comprobar si otra restricción se puede derivar de las anteriores.

1.5. Objetivos

Como se mencionó anteriormente, el objetivo principal de este trabajo es desarrollar un resolutor de restricciones que permita usar información del contexto para aplicar ciertas simplificaciones en el proyecto `FORVES` que necesitan de ciertas condiciones previas para ser usadas. Además, al igual que en el resto del proyecto, dicho resolutor de restricciones debe estar verificado en Coq. Para ello elegimos los siguientes subobjetivos.

- Estudiar Coq como herramienta para el desarrollo de *software* verificado.
- Modelizar el problema en Coq.
- Implementar un comprobador de implicaciones que sea capaz de determinar si una restricción puede ser derivada de información de contexto.
- Verificar el comprobador de implicaciones en Coq.

1.6. Plan de trabajo

Cada objetivo a su vez se dividió en distintas tareas y cada una de estas tareas se distribuyo en el periodo de tiempo asignado al proyecto.

1. Estudiar Coq como herramienta para el desarrollo de *software* verificado.
    - 1.1. Leer *Logical Foundations* [9] hasta el Capítulo 6 - *Logic*.
    - 1.2. Terminar *Logical Foundations* [9].
2. Modelizar el problema en Coq.
    - 2.1. Definir la estructura de las restricciones relevantes.
    - 2.2. Definir la noción de implicación de restricciones.
    - 2.3. Definir qué es un comprobador de implicaciones.
3. Implementar un comprobador de implicaciones que sea capaz de determinar si una restricción puede ser derivada de información de contexto.

    3.1. Implementar una versión básica de un comprobador de implicaciones, para asegurarse de la definición es la deseada.
    3.2. Implementar una versión más potente del comprobador de implicaciones usando un resolutor de restricciones.

4. Verificar el comprobador de implicaciones en Coq.
    4.1. Desarrollar la teoría para verificar el comprobador de implicaciones básico.
    4.2. Desarrollar la teoría para verificar el comprobador de implicaciones definitivo.

El periodo asignado para el trabajo fueron los meses de enero a mayo de 2024. Los dos primeros meses se reservaron para familiarizarse con las herramientas a utilizar, como Coq o `FORVES2`. El mes siguiente se dedicó a modelizar el problema en Coq, decidiendo y estabilizando la versión final de las definiciones que se iban a usar e implementando la versión básica del comprobador de implicaciones para ver como estas definiciones interactuarían con el resto del proyecto. Durante este mes se consideraron varios algoritmos para implementar el resolutor de restricciones. La implementación del comprobador de restricciones definitivo se dejó para mayo, cuando se implementó el algoritmo en Coq y se desarrolló la teoría necesaria para verificarlo. Esta teoría se acabó implementando de forma independiente al resto de definiciones del proyecto, y pese a que originalmente consideramos incluir la integración de esta teoría con el resto del proyecto como una subtarea adicional, rápidamente nos dimos cuenta de que este esfuerzo estaba fuera del ámbito del trabajo. Para más información ver la Sección 5.2.

El ciclo de desarrollo final se puede visualizar como diagrama de Gantt en la Figura 1.1.

El trabajo se desarrolla en el repositorio de GitHub[10] del proyecto `FORVES2`. Los archivos relevantes al trabajo son `constraints.v` y `octagon.v`, donde el primero usa algunos términos definicios en otros archivos del proyecto.

[10] `https://github.com/costa-group/forves2`

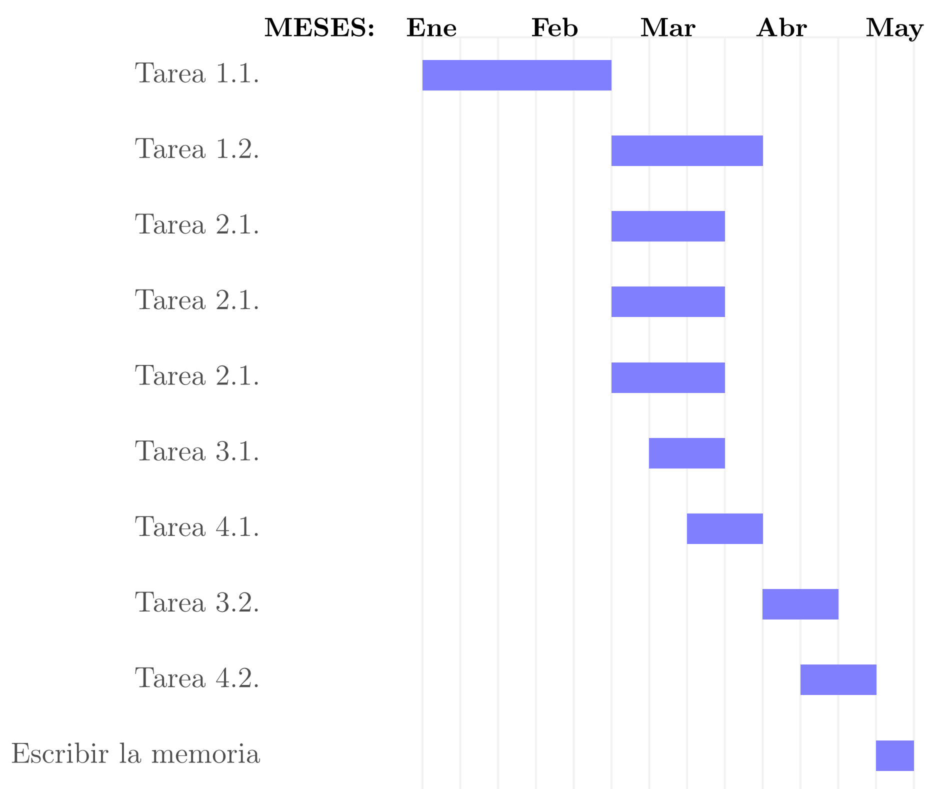


Figure 1.1: Diagrama de Gantt del trabajo.

Chapter 2

State of the art

This chapter is divided in three sections, each touching upon a specific technology of interest for this project. In the first section we give an overview of blockchain technology and smart contracts, focusing on Ethereum's implementation. We also touch upon the topic of EVM optimization and the `GASOL` project. In the next section we introduce the concept of "octagonal constrainst" and give further background in relevant results of its field which will be of use in Section 4.1. In the last section we give an in-depth overview of the Coq proof assistant.

2.1. Blockchain

A blockchain implements a distributed ledger, a database of transactions between entities which is not hosted by any central authority but shared among multiple computers in a network. The name comes from the mechanism in which these transactions are stored: groups of transactions called "blocks" are linked together forming a "chain". Once a block of transactions has been recorded in the block-chain, no further changes can be applied to it without needing to change all subsequent blocks, which in real world scenarios is made to be unfeasible, thus ensuring the immutability and transparency of the data.

Blockchain technology has diverse applications, the most famous one being as a form of currency. One of these use cases of particular interest is implementing smart contracts. Smart contracts are self-executing contracts whose terms are directly expressed as code. They run on a blockchain network and automatically execute actions when predefined conditions are met, without the need to rely on a trusted intermediary since their execution is verified by the decentralized network. The usage of smart contracts has enabled the development of new and innovative decentralized applications (DApps).

2.1.1. The Ethereum Virtual Machine

Ethereum is a prominent blockchain platform which stands out for its robust support for smart contracts through the Ethereum Virtual Machine (EVM). The EVM is a runtime environment that enables the execution of code written in high-

level languages like Solidity. Solidity code is compiled into EVM bytecode, which is composed of instructions operating on a stack-based virtual machine. Each operation performed in the EVM incurs a cost measured in gas, Ethereum's unit of computational cost. This cost is covered by the user interacting with the smart contract and is payed in the Ethereum blockchain's native currency, of the same name. This serves as a mechanism to prioritize and meter the execution of code in the network, which ensures that the network remains efficient and secure, by encouraging developers to be mindful of their use of resources. This contributes to preventing spam attacks and resource abuse, ensuring the stability and sustainability of the Ethereum ecosystem.

As stated in Ethereum's Yellow Paper [13] the EVM is a stack-based machine virtual machine with a memory model. The word size of the machine is 256-bits, so 256-bits values are the maximum which can be handled by the stack and virtual machine. When a smart contract is to be executed, a new stack of up-to 1024 words is created for its execution, which will be destroyed after the execution has completed. The main purpose of the stack is to provide the arguments to bytecode instructions and host temporary values. For more long-lived values the EVM provides two solutions: the *memory* for local variables and the *storage* for global variables. The memory is a word-addressed byte array used to store volatile, while the storage is a global memory which persists its state between smart contract calls. The storage's state is part of the state of the blockchain, while the memory is unique to the machine which executes the contract.

Therefore, for a change to be persisted between executions it needs to be recorded in memory.

EVM instructions modify the state of the EVM in two main ways:

- Manipulating the values in the stack.
  For example, consider the `ADD` instruction which pushes to the top of the stack the addition of its first two elements. When simulating the execution of a program, this will be represented by showing the state of the stack before and after the execution of the instruction.

  ```
  [x0,x1]
   ↓ ADD
  [x0+x1]
  ```

- Producing an effect.
  An *effect* is any change which can be observed outside of the stack. Both writing to memory and storage are considered as effects. For example, the `MSTORE` instruction pops the first two elements of the stack and stores the value of the second element in the memory offset referenced by the first element. When simulating the execution of a program, we represent effects, if present, by putting them in parenthesis next to the state of the stack.

  ```
  [x0,x1]
   ↓ MSTORE
  []    (MEM[x0] = x1)
  ```

The EVM defines many more instructions. For the purposes of this chapter, only a few more will be described.

- SWAP1 exchanges the top element in the stack with the second element bellow it.

```
[x0,x1]
 ↓ SWAP1
[x1,x0]
```

- SWAP2 exchanges the top element in the stack with the second element bellow it.

```
[x0,x1,x2]
 ↓ SWAP2
[x2,x1,x0]
```

- SWAP3 exchanges the top element in the stack with the third element bellow it.

```
[x0,x1,x2,x3]
 ↓ SWAP3
[x3,x1,x2,x0]
```

- POP discards the top element of the stack.

```
[x0]
 ↓ POP
[]
```

Consider the following bytecode sequences in Listings 2.1 and 2.2.

```
SWAP3 SWAP1 SWAP2 MSTORE SWAP1 SWAP2 ADD SWAP1 MSTORE
```

Listing 2.1: EVM bytecode sequence 1

```
MSTORE MSTORE POP
```

Listing 2.2: EVM bytecode sequence 2

With the previously defined format for representing EVM bytecode execution, we can represent the effects of executing both of these programs. First, consider the execution of Listing 2.1.

```
[x0,x1,x2,x3,x4,x5]
 ↓ SWAP3
[x3,x1,x2,x0,x4,x5]
 ↓ SWAP1
```

```
[x1,x3,x2,x0,x4,x5]
 ↓ SWAP2
[x2,x3,x1,x0,x4,x5]
 ↓ MSTORE
[x1,x0,x4,x5] (MEM[x2] = x3)
 ↓ SWAP1
[x0,x1,x4,x5] (MEM[x2] = x3)
 ↓ SWAP2
[x4,x1,x0,x5] (MEM[x2] = x3)
 ↓ ADD
[x4+x1,x0,x5] (MEM[x2] = x3)
 ↓ SWAP1
[x0,x4+x1,x5] (MEM[x2] = x3)
 ↓ MSTORE
[x5]          (MEM[x2] = x3; MEM[x0] = x4+x1)
```

Finally, consider the execution of the Listing 2.2

```
[x0,x1,x2,x3,x4,x5]
 ↓ MSTORE
[x2,x3,x4,x5] (MEM[x0] = x1)
 ↓ MSTORE
[x4,x5]       (MEM[x0] = x1; MEM[x2] = x3)
 ↓ POP
[x5]          (MEM[x0] = x1; MEM[x2] = x3)
```

Both listings produce different effects upon execution. However, under some special circumstances, which will be explored in the next section, both listings can be shown to be equivalent. Under these circumstances, Listing 2.2 would be preferable over 2.1 since it uses less and cheaper instructions.

2.1.2. Optimization of smart contracts

The GASOL project implements a superoptimizer for EVM bytecode. Superoptimization is a compilation technique that searches, for a given jump-free sequence of instructions, a semantically equivalent sequence of instructions which is optimal by some metric, like memory usage or execution cost. Since the superoptimizer requires these sequences of instructions to not have bifurcations, to optimize a whole program it first extracts all the sequences of instructions which do not perform jumps and optimizes those separately before reassembling them back together. In doing so, it remembers which conditions triggered those jumps so it can gain more information on which states of the program are possible for each section.

The FORVES project, although not dependent on GASOL, was originally started to certify that its optimizations did not modify the semantic of the given programs. That is why FORVES inherited the requirement that the sequences of instructions be jump-free. The currently implemented FORVES verifier has been thoroughly tested on outputs of GASOL, but some optimizations prove harder to verify than others.

For instance, the previously presented Listings 2.1 and 2.2 are unoptimized and optimized versions of the same jump-free sequence of EVM bytecode instructions. While the end result of the stacks are equivalent for both executions, we cannot ensure that the effects they produce are equivalent. For starters, we cannot ensure that the memory accesses can be commuted. After all, if `x0` = `x2` = 0 then after the first program we would have `MEM[0] = x4+x1` but after the second we would have `MEM[0] = x3`, which need not be the same if `x3` ≠ `x4+x1`. Furthermore, we would need that `x4+x1` = `x1`, which is only possible if `x4` = 0.

However, some of these conditions may be fulfilled if we consider the contextual information. For example, consider the following contextual information, a list of added constraints to the previous variables.

- `x4` = `x5`
- `x5` = 0
- `x0` ≥ `x2` +128

From the last constraint we can derive that the memory accesses to the offsets referenced by `x0` and `x2` are disjoint, since the word size of the EVM is of 32 bytes and $32 \leq 128$. Therefore we can reorder both writes while preserving the semantics of the code. Finally, from the first two constraints we can derive that `x4` = 0 which ensures that `x4` + `x1` is in fact equivalent to `x1`.

This example motivates the necessity of taking contextual information into account when verifying optimizations.

2.2. Octagonal constraints

The constraints we are interested on can be reduced to a special type of constraint called octagonal constraints. Octagonal constraints are constraints of the type $\pm x \pm y \leq d$ or $\pm x \leq d$ where x and y are integer variables. In general we assume that d is an integer, since otherwise we could exploit the fact that the left hand side of both types of constraints result in integers to derive a more strict inequality using $\lfloor d \rfloor$ instead. This new constraint $\pm x \pm y \leq \lfloor d \rfloor$ is called the *tightened* constraint of $\pm x \pm y \leq d$.

Octagonal constraints have been studied in depth in the literature. In particular, efficient algorithms [12, 10] have been derived to compute the transitive closure of a set of constraints. The transitive closure of a set of constraints C is a set of constraints C' where for every pair of variables x and y defined in the constraints of C we have a d such that the constraint $x + y \leq d$ is in C' and a d' such that $x \leq d'$ is in C'. Furthermore, the choices of d and d' are the smallest possible such that they can still be derived from the constraints in C.

In [10] an $O(n^3 \log n)$ algorithm is described to compute the tightened transitive closure of a set of octagonal constraints where n is the number of integer variables. The same article cites [12] for providing an $O(n^4)$ algorithm which relies on the application of successive rules to iteratively construct the transitive closure.

2.3. The Coq Proof Assistant

From Coq's website[1]

> Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the certification of properties of programming languages, the formalization of mathematics, and teaching.

Of the mentioned typical applications in this project we are most interested in the first, the "certification of properties of programming languages". In particular we are interested in verifying that certain algorithms satisfy certain properties that ensure they are implemented correctly and fulfill their purpose. The Coq proof assistant has been used extensively to verify a multitude of projects.

- `CompCert` [7], a certified compiler for the majority of the C language.
- `Iris` [1], a higher-order concurrent separation logic framework, used for reasoning about safety of concurrent programs, as the logic in logical relations, to reason about type-systems, data-abstraction, ... It has been used in other projects such as RustBelt [2].
- `Fiat-Crypto` [6], a tool for cryptographic primitive code generation.
- `ConCert` [3], a framework for smart contract verification in Coq.
- `Cosette` [4], an automated prover for checking equivalences of SQL queries.

In essence, it is a tool for writing and verifying mathematical proofs and developing certified software. To provide this functions, Coq provides a way of creating programs, functioning as a programming languages, and a way to develop proofs, functioning as a proof assistant.

2.3.1. Coq as a functional programming language

Although, at its core, Coq is a typed functional programming language, we interact with this language in a peculiar way. When you write a program in Coq you can think of it as having a conversation with the language, where each statement is a question and its result is what Coq gives you as an answer. It is not that different from writing your programs in a REPL in other languages. However, Coq allows you to be very creative with your questions. For example, the following command will prompt Coq to show the internal representation of its grammar. Pay attention to the final dot. All Coq commands must end in a dot.

```
Print Grammar constr.
```

[1] `https://coq.inria.fr/`

```
Entry constr is
[ LEFTA
  [ "@"; global; univ_annot
  | term LEVEL "8" ] ]
and lconstr is
  ...
```

At first this might come off as odd. Why would one need such a command? The answer is that, while in most programming languages we regard the grammar as something static, Coq allows you to alter the way it parses your definitions. Therefore, it is useful to know what Coq currently thinks the grammar looks like.

However, these commands are not to what we refer to when we talk about Coq being a programming language. The programming language inside Coq can be accessed through some special commands, such as `Inductive` or `Definition`.

`Inductive` allows us to define data types. They work similarly to enumerations in other languages. Consider, for example, the following definition of the booleans as a type containing two constructors, `true` and `false`.

```
Inductive bool :=
  | true
  | false.
```

An inductive type defined in this way can have any number of constructors, including zero!

```
Inductive void := .
```

You can check for the type of a value with the `Check` command.

```
Check false.
```

```
false
     : bool
```

While the `Inductive` command allows us to define types, `Definition` allows us to define values. This works in a similar way to general programming languages. The following statement assigns to the variable named `my_true` the value `true`.

```
Definition my_true := true.
```

A function can be defined through the `Definition` command by specifying its arguments on the left side of the := symbol. The parenthesis around the arguments are only needed if you want to specify the argument's type, or if the type itself could not be inferred by Coq itself. For example we define the function `always_true` which takes a boolean argument `b` and always returns `true`.

```
Definition always_true (b: bool) := true.
```

To express the return type of a function we also use the `:type` syntax, putting it right before the := symbol. The following line defines the identity function for values of type `bool`.

```
Definition bool_identity (b: bool) : bool := b.
```

To the right side of the := symbol we can put any Coq expression. Technically, the language of expressions in Coq is called Gallina but people usually refer to it as just Coq informally. Calling a function is an expression, and the syntax used is the same as when defining one. To illustrate this we introduce the `Compute` command, which prompts Coq to evaluate an expression and show its result.

```
Compute bool_identity true.
```

```
= true
     : bool
```

A useful construct of Gallina is that of the `match _ with ... end` which lets us act differently depending on the way values were constructed. This can be used, for example, to define the boolean `notb` function, which exchanges `true` for `false` and vice versa.

```
Definition notb b := match b with
                     | true => false
                     | false => true
                     end.
```

Observe that in this case we could omit the parenthesis in the argument definition, since Coq was able to infer the type of `b` from how it was used in the expression.

We can also create functions which return anonymous functions by using the `fun` keyword. The arguments are specified with the same syntax as before, with the main difference being that now the => symbol is used to separate the arguments from the body, and not :=. The types of functions use the -> symbol to separate the argument type from the return type. Consider the following function which implements a curried version of the boolean and function.

```
Definition andb (b: bool) : bool -> bool :=
  fun b' => match b with
            | true => b'
            | false => false
            end.
```

As with most functional programming languages, functions in Coq can be curried, which means that they can be partially applied, providing only one argument and returning a function which takes the others. Therefore this type of definition is, in fact, unnecessary. We could have achieved the same result by having the function take both arguments `b` and `b'` directly.

When defining a new type with `Inductive`, the constructors themselves can also take arguments. This works as the main way of storing values in Coq, similar to how records work in other programming languages. For example, this is how we would define a named pair of two boolean values.

```
Inductive two_bools := my_bools (b b': bool).
```

Notice that `b'` is a valid identifier in Coq. While this is not unique to Coq (OCaml also has this feature), it is still novel enough to be worth pointing it out. Also notice that since both `b` and `b'` had the same type we could specify it under the same set of parenthesis.

`Inductive` definitions are also allowed to be recursive, as long as Coq is able to prove that this recursion comes to an end. In general this means that at least one constructor is not recursive. For example, here is the definition of Peano natural numbers in Coq.

```
Inductive nat :=
  | O
  | S (n: nat).
```

You can also define recursive functions, but not through the `Definition` command. We use the `Fixpoint` command instead. When using `match` over a type with arguments in its constructors, it can be useful to use an identifier instead of a value in order to bind the value to the identifier. Observe how this is used in the following definition of addition for Peano's natural numbers.

```
Fixpoint add (n m: nat) :=
  match n with
  | O => m
  | S n' => add n' (S m)
  end.
```

If the argument itself will not be used, you can also use the special identifier `_` to discard it, like it is used in the following example of the function `is_zero`.

```
Fixpoint is_zero(n: nat): bool :=
  match n with
  | O => true
  | _ => false
  end.
```

`Fixpoint` functions in Coq also have the catch that Coq must be able to ensure that they are terminating. Therefore, a function that loops infinitely cannot be defined in Coq.

```
Fixpoint loop (n: nat): nat := loop n.
```

```
Recursive definition of loop is ill-formed.
In environment
loop : nat -> nat
n : nat
Recursive call to loop has principal argument equal to
"n" instead of a subterm of "n".
Recursive definition is: "fun n : nat => loop n".
```

These are the basic building blocks of Coq's programming language. There are not any built-in data types commonly found in other languages like booleans or numbers. These are instead defined as part of the standard library, in `Coq.Bool.Bool` and `Coq.Init.Nat`. Common expressions such as the `if` expression are actually just syntactic sugar over the `match` expression. The same is true for a `let ... in` expression, which lets you bind an expression to a variable and use it in another expression.

```
let x := 10 in x + 1
```

In Coq, this statement desugars to a `match` expression.

```
match 10 with
| x => x + 1
end
```

To let Coq know that we want to use the definitions for `bool` and `nat` defined in its standard library, we would use the following commands.

```
From Coq Require Import Bool.Bool.
From Coq Require Import Init.Nat.
```

Besides including the definitions mentioned before, the standard library also defines some common functions, such as addition, and provide some syntactic sugar to work with them. For example, the two following statements are equivalent since they make use of the same function.

```
Compute add 1 2.
```

```
= 3
     : nat
```

```
Compute 1 + 2.
```

```
= 3
     : nat
```

To show the syntactic sugar that Coq is currently aware of you can use the following command.

```
Print Visibility.
```

```
...
Visible in scope nat_scope
"x >= y" := (ge x y)
"x > y" := (gt x y)
"x <= y <= z" := (and (le x y) (le y z))
"x <= y < z" := (and (le x y) (lt y z))
"n <= m" := (le n m)
"x < y <= z" := (and (lt x y) (le y z))
```

```
"x < y < z" := (and (lt x y) (lt y z))
"x < y" := (lt x y)
"x - y" := (Nat.sub x y)
"x + y" := (Nat.add x y)
"x * y" := (Nat.mul x y)
```

One thing to point out from the output of this command is the fact that boolean equality of two numbers, recognized as the function `eqb`, is not expressed as == or = but with the =? symbol. This is to highlight that this operator returns a boolean value, since unlike other programming languages, booleans are not built-in into the language. Furthermore, the = operator is reserved for some other purpose in Coq, which we will explore in the next section.

Coq also allows overloading a symbol to different operations depending on the types of its operands. This can ease the readability of some proofs and theorems, however it can also sometimes lead to ambiguity. To prevent this, Coq offers the possibility to define this syntactic sugar under a scope, and later specify under which scope a given piece of syntax should be interpreted by. This is done by enclosing the syntax in parenthesis and following it with the \% symbol and the name of the scope, which usually tends to correspond to a type.

This is useful, for example, when juggling with multiple definitions of numbers. Besides `nat`, Coq also defines the data-types `N` and `Z` for naturals and integers. The differences between `nat` and `N` lie in how they are defined. `nat` implements the Peano natural numbers, encoded as zero and its successors. `N` on the other hand follows the computer representation of numbers, using a sequence of zeroes and ones. Fortunately, we can use the + symbol to express addition for both data-types.

```
Check (1 + 1)%nat.
```

```
(1 + 1)%N
     : N
```

```
Check (1 + 1)%N.
```

```
(1 + 1)%nat
     : nat
```

2.3.2. Coq as a theorem prover

Besides the commands that allow you to interface with the programming language, Coq also offers commands with which you can define theorems and prove them. In fact, the commands `Theorem`, `Lemma` and `Example`, the main way in which you interact with the theorem proving side of Coq, are all equivalent with one another. The different names are simply for the developers to classify the properties they are trying to prove as they see fit.

Consider the following proposition. Pay attention to how we now use the : symbol to separate the proposition with its name instead of the := symbol as we did in the programming language.

```
Example true_is_true : true = true.
```

The proposition appears after the first : symbol, and it follows a similar syntax to expressions. After this command is executed Coq registers the proposition you want to prove and awaits for you to provide a proof of this proposition. We do this by entering proof mode using the `Proof` command.

In proof mode, Coq shows you the proposition you have to prove as a goal. In graphical interfaces this is usually the proposition which appears under the horizontal line.

```
1 subgoal


======================== (1 / 1)

true = true
```

The space above the horizontal line is reserved for local definitions, which are definitions that can only be accessed from within the proof. Most of the time these definitions refer to our hypothesis. There are no hypothesis at the moment, which is why the only text above the horizontal line is the goal counter. The goal is the current proposition we need to prove. There can be more than one goal to prove in a proof, and we say that the proof has concluded when there are no goals left.

Inside proof mode, a new set of commands a syntax are available to us to construct the proof. These are called *tactics*. For example, to prove the goal `true = true` we wouldd use the `reflexivity` tactic, which concludes an equality if both sides are syntactically the same. This is exactly the case for this goal, which has `true` on both sides.

Once the goal has been proven you can exit proof mode using the `Qed` command. Therefore, a theorem and its proof would look like this

```
Example true_is_true : true = true.
Proof.
  reflexivity.
Qed.
```

This example is not terribly exciting. It would be more interesting if we could prove a property for a greater amount of values, not just one. To achieve this in Coq we use the `forall` keyword, which has a similar syntax to that of function application.

```
Example bool_refl' : forall b : bool, b = b.
```

If we enter proof mode, the goal is now `forall b : bool, b = b`. We cannot conclude this proof with `reflexivity`, since the goal is not an equality, it has a quantifier beforehand. What we would like to say is that the proof is the same

regardless of our choice of `b`. In Coq we express this by introducing an arbitrary value `b` and proving the theorem for that `b`. This is done through the `intros` tactic.

```
Proof.
  intros b.
  reflexivity.
Qed.
```

Proving something for all elements of a type is pretty powerful, but usually we are only interested in proving things for elements which satisfy certain properties. To do this we make use of implication, written with the `->` symbol.

```
Example bool_eq_sym' : forall b b' : bool, b = b' -> b' = b.
```

To start the proof we introduce the variables `b` and `b'` with `intros` as before. This leaves us in the following state.

```
1 subgoal

b, b' : bool

========================= (1 / 1)

b = b' -> b' = b
```

To proceed, we use `intros` again, and we give the new constraint the name `eq_b_b'` to reflect its meaning

```
1 subgoal

b, b' : bool
eq_b_b' : b = b'

========================= (1 / 1)

b' = b
```

What we have introduced this time under the name `eq_b_b'` is not a value, but the evidence that the proposition `eq_b_b'` is true. We can later use that evidence with other tactics, like `rewrite`, which takes an equality and rewrites all appearances of the expression on the left side of the = symbol with that on the right side of the equality.

```
1 subgoal

b, b' : bool
eq_b_b' : b = b'
```

```
========================= (1 / 1)

b' = b'
```

At this point we can conclude the proof with the `reflexivity` tactic.

```
Example bool_eq_sym' : forall b b' : bool, b = b' -> b' = b.
Proof.
  intros b b'.
  intros eq_b_b'.
  rewrite eq_b_b'.
  reflexivity.
Qed.
```

The dual of the universal quantifier `forall` is the existential quantifier `exists`, which states that there is a value of one type which satisfies the specified proposition.

```
Example has_zero : exists n: nat, n = 0.
```

When proving a goal with an `exists`, Coq expects you to provide a value with the equally-named `exists` tactic, and show that the property holds for the given value.

```
Proof.
  (* Goal is [exists n : nat, n = 0.] *)
  exists 0.
  (* Goal is [0 = 0.] *)
  reflexivity.
Qed.
```

In these proofs so far, we have treated all elements of a type the same way, not making a distinction on how they were constructed. However, for some proofs it is necessary to make this distinction.

```
Theorem negb_involutive : forall b : bool, negb (negb b) = b.
```

To prove this statement, we need to show that it holds true for both `true` and `false`. To do so we use the `destruct` tactic, which takes a term as an argument and creates as many subgoals as constructors it has. In each of these subgoals, the term has been replaced by one of its constructors. Therefore, `destruct` allows us to prove that a proposition is true depending on how the term was constructed.

In this case, the first subgoal asks us to prove `negb (negb true) = true`. We can use the `simpl` tactic to ask Coq to simplify the goal, for example by executing the `negb` function through its definition. This leaves us with `true = true`, which we can prove through the `reflexivity` tactic. The same argument holds for the second subgoal

```
Proof.
  intros b. destruct b.
  - simpl.
    reflexivity.
  - simpl.
    reflexivity.
Qed.
```

As a matter of fact, the `true = true` goal is something we already proved in the `true_is_true` example before. We could reuse this proof by using the `apply` tactic instead, which receives the name of a proposition previously defined and attempts to fit it into the current goal, by trying to infer the values of quantified variables. For example, we could have also used the `bool_eq_refl'` theorem, and Coq would have set the value of `b` to `true`.

The `apply` tactic also works if the proposition to apply is part of an implication. If we have a lemma `h: P -> Q` and we need to prove `Q`, we can use `apply h`, which will change the goal to `P`, since if we are able to prove `P` then we can prove `Q` by using `h`. An example of such a situation would be using the previously defined `bool_eq_sym'`.

```
Lemma  h': forall b: bool, b = true -> true = b.
Proof.
  intros b b_true.
  (* Goal is [true = b] *)
  apply bool_eq_sym'.
  (* Goal is [b = true] *)
  apply b_true.
Qed.
```

Notice how we conclude this example by applying a proof which lives as a local definition. The apply tactic is not restricted to the theorems available to Coq, but also to those arising from the context of the proof.

Negation in Coq is not a first-class feature, but it is implemented in terms of implication and `False`. The `False` proposition represents a proposition which cannot be proven, its dual is the `True` proposition, for which there is one trivial proof called `I`.

Then, the negation of a proposition `P`, written as `~ P`, is defined as `P -> False`. The idea behind it is that if having a proof of `P` allows you to derive a proof of `False` then you could not derive a proof for `P` in the first place, since there is no proof for `P`.

The same way we have `reflexivity` to end a proof if the goal is an equality and both sides of the equality are constructed with the same constructors, the `discriminate` tactic allows us to end a proof if given one hypothesis of an equality which can be shown to be false since both sides use different constructors.

For instance, consider the following proposition.

```
Example one_not_zero : ~ (1 = 0).
Proof.
```

```
  unfold not. (* Transforms [~ (1 = 0)] into [(1 = 0) -> False] *)
  intros h.   (* Introduces hypothesis [h: 1 = 0] *)
  discriminate h. (* Concludes that hypothesis h cannot be obtained *)
Qed.
```

We have defined `~ P` to be `P -> False`. If we revert the terms in that implication we have `False -> P` which is trivially true for any proposition `P`. This fact is known in the literature as the principle of explosion or *ex falso quodlibet*. In Coq this principle shows in some proofs where instead of proving the goal it may be convenient that with the assumed hypothesis we can derive a proof for false. This principle is accessed through the `exfalso` tactic.

```
Theorem not_true_is_false' : forall b : bool,
  ~ (b = true) → b = false.
Proof.
  intros b not_b_true.
  destruct b.
  - (* b = true *)
    unfold not in not_b_true.
    exfalso.
    apply not_b_true.
    reflexivity.
  - (* b = false *)
    reflexivity.
Qed.
```

Besides negating a proposition, Coq also allows us to use the usual logical connectives. And is expressed with the symbol /\ and or with the symbol \/, mimicking their shape's in mathematical notation: $\wedge$ and $\vee$.

When dealing with /\ in the goal, we can use the `split` tactic to divide the goal into two, which we will have to prove separately. If the /\ is instead in a hypothesis `h`, we can use the `destruct h` tactic, which would separate both sides of the conjunction in different hypothesis. Coq will try to give appropriate names to the new hypothesis, but most of the time it is a better idea to give the names on your own. To do so, you can use `destruct h as [hl hr]`, where `hl` and `hr` are the names given to the left and right sides of the conjunction respectively.

When dealing with \/ in the goal, we need to decide which part of the proposition we would like to prove in order to prove the hypothesis. To choose the left side of the disjunction we use the appropriately named `left` tactic, with the `right` tactic choosing the right side of the disjunction as expected. If the \/ is instead in a hypothesis `h`, we can again use the `destruct h` tactic, which will ask us to prove the same goal twice: once with the left side of the disjunction `h` and once with the right side of the disjunction `h`. In both of these options the sides of the disjunction would replace the `h` hypothesis, but it is also possible to rename them by employing the tactic as `destruct h as [hl | hr]`, where `hl` and `hr` are again the names of the left and right side of the disjunction respectively.

Before moving on to the next topic, I wanted to point out a particularity of Coq's mathematical foundations. Notice that when handling a goal of type $P \vee Q$ we need to know which between P or Q holds before proceeding with the proof. This is due to Coq's logic being constructive. In a non-constructive logic proving that a proposition cannot be false is equivalent to proving that that proposition is true. In a constructive logic, however, a proposition is only true if you show it to be true. While nuanced, this is a powerful distinction.

Consider the following proposition:

There exist irrational numbers a and b such that a^b is rational.

In classical logic, a proof of the previous would proceed like this. We assume we have already proven that $\sqrt{2}$ is irrational. Then consider whether $\sqrt{2}^{\sqrt{2}}$ is rational. If it is, we can conclude the proof with $a = b = \sqrt{2}$. Otherwise $\sqrt{2}^{\sqrt{2}}$ is irrational. We choose $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$$

Since 2 is rational this concludes the proof.

Notice how we have concluded the proof without knowing the values of a and b. This happened because we were able to assume the following:

$$\sqrt{2}^{\sqrt{2}} \text{ is rational } \vee \sqrt{2}^{\sqrt{2}} \text{ is irrational.}$$

A constructive logic has the advantage that, if something is proven within it, you are guaranteed to have values for what was proven. Nevertheless, sometimes proofs require of the `excluded_middle`. For those cases, Coq offers `excluded_middle` as an Axiom in the module `Coq.Logic.ClassicalFacts`. An axiom is a proposition which Coq accepts as true without proof, and can be declared with the `Axiom` command. However, it is dangerous to do so, one may introduce a proposition which would enable proving `true = false`. Fortunately, we can rely on Coq's provided assumptions to not break the consistency of its logic.

2.3.3. The Curry-Howard correspondence

In the field of programming languages and type theory, the Curry-Howard correspondence states that there exists an isomorphism between programs and proofs, and types and propositions. So far we have treated Coq the programming language and Coq the proof assistant as two different things, but in fact, they are the same. Whenever we wrote `Example` or `Theorem`, we could have used `Definition` instead!

```
Definition true_is_true' : true = true.
Proof.
  reflexivity.
Qed.
```

The type of `true_is_true` is in fact the proposition we are trying to prove, and its value is the proof built in proof mode. In fact, not even proof mode was necessary. We could have specified the "proof object" directly after the := symbol. This is made apparent if we ask Coq to print one of our prior proofs.

```
Print has_zero.
```

```
has_zero =
ex_intro (fun n : nat => n = 0) 0 eq_refl
     : exists n : nat, n = 0
```

Here we see that we construct a proof value using the `ex_intro` constructor, with the following parameters:

- `fun n : nat => n = 0` is a function which takes a `nat` and returns the proposition `n = 0`
- 0 is the witness of the `exists n : nat, n = 0`, that is, the value of `n` for which the property is true
- `eq_refl` is the proof that `0 = 0`. This is what gets applied when we use the `reflexivity` tactic!

The feature which allows us to use Coq as a theorem prover is its type system. In fact, Coq's type system is not like those of other general purpose language, since it allows the definition of dependent types. A dependent type is a type whose definition depends on a value. This allows us to make types such as `Vector.t nat 10`, of all lists of natural numbers of length 10, or more importantly, since propositions are types it allows us to refer to individual values in our propositions, instead of just their types.

Dependent data types can be created with the `Record` command.

```
Record dependent_example := {
  a: nat;
  proof_a_not_0: a <> 0
}.
```

The `dependent_example` type represents all pairs of naturals `a` and proofs that `a` is not 0. We can generalize this example to define the subset of elements of a type A which fulfill some condition P, that is $\{x \in A : P(x)\}$. This type of record is defined in Coq's standard library and it is called `sig`.

```
Print sig.
```

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> {x : A | P x}.

Arguments sig [A]%type_scope P%type_scope
Arguments exist [A]%type_scope P%function_scope x _
```

`sig` is usually used through a special syntax also defined in the standard library where `{x : A | P x}` represents the type `sig A P`.

The `forall` keyword behaves like function application, with the difference that it allows the return type to use the argument's value, that is, it allows the return type to depend on the function's parameters. Otherwise, `forall n : nat, nat` is the same as `nat -> nat`, as can be seen in the following example.

```
Definition a : forall n: nat, nat := add 1.
Check a.
```

```
a
     : nat -> nat
```

The `True` and `False` properties are in fact defined with `Inductive`. The definition of `False` is equivalent to our previous definition of `void`, an inductive type with no constructors, and `True` is an inductive type with one constructor, `I`. Intuitively it makes sense, since there is not proof of falsehood, therefore it has no constructors, and you can always prove `True` without needing any extra information, hence the constructor without arguments.

Dependent types also allow us to achieve something similar to parametric polymorphism. Consider the previously defined `bool_identity function`. This function is a less powerful version of the generic `identity` function which works for arguments of any type. These types of functions are defined in other functional programming languages through the use of parametric polymorphism, making both the input and output types generic. However, since in Coq types are first class values we do not need parametric polymorphism to define such a function.

```
Definition identity' (A: Type)(b: A) := b.
Compute identity' bool true.
```

This definition is possible since we are allowed to make the type of the second argument depend on the value of the first with dependent types. However, using this identity function is noticeably less ergonomic than its counterpart in other functional programming languages since we also need to explicitly specify the type of its argument. In other functional programming languages, this task is left to the type checker through the feature of type inference. Instead, Coq introduces a more powerful feature: implicit arguments. An explicit argument is defined by surrounding it with curly brackets instead of parenthesis.

```
Definition identity {A: Type}(b: A) := b.
Compute identity true.
```

Coq will attempt to derive the value for implicit arguments automatically, using globally available information as well as its signature. You can also ask Coq to infer the value of a parameter which was not marked as implicit by using the _ identifier instead of its value.

```
Compute identity' _ true.
```

To provide all arguments of a function explicitly, even if they were defined as implicit, you can refer to the function with a prepended `@`.

```
Compute @identity bool true.
```

This is occasionally useful if Coq is struggling to infer the implicit argument[2].

In other programming languages, many commonly used functions are made available in the form of a standard library. Coq does the same thing, with its standard library perhaps containing more basic definitions than in most other functional programming languages since the booleans and integers are delegated to it. For example, Coq's standard library defines the `list` data type and its constructors, and the usual operations over them such as `map`, `filter`, `fold_left` and `fold_right` or `forallb` and `existsb`. These are defined in the `Coq.Lists.List` module.

```
Print existsb.
```

```
existsb =
fun (A : Type) (f : A -> bool) =>
fix existsb (l : list A) : bool :=
  match l with
  | nil => false
  | a :: l0 => f a || existsb l0
  end
     : forall A : Type, (A -> bool) -> list A -> bool
```

However, besides providing data types and operations Coq also provides theorems which let us reason about these data types and operations. For example, in `Coq.Lists.List` the `existsb_exists` lemma is defined, which establishes the relationship between the `existsb` function and the `exists` quantifier.

```
Print existsb.
```

```
existsb_exists
     : forall (A : Type) (f : A -> bool) (l : list A),
       existsb f l = true <-> (exists x : A, In x l /\ f x = true)
```

These lemmas are used extensively throughout this project, and they are also the reason why we try to define things in terms of functions in the standard library whenever possible.

[2]Coq's algorithm for type checking and type synthesis can be exponential in time in the worst case. See [8] for more information.

Chapter 3

Modelization of the problem in Coq

The purpose of FORVES is to certify that two sequences of EVM instructions are semantically equivalent. To do so, we need to define what we mean by semantic equivalence.

3.1. Semantic equivalence of programs

We can represent a program as a function which transform state where by state we mean an assignment of variables to values. In generic programming languages state is usually visualized with a map or a dictionary.

```
state = {
    "x": 10,
    "y": 3,
}
```

The reason we use maps to represent these objects is that they are memory efficient, since their cost in efficiency is proportional to the number of elements they assign. However, the canonical way in which we represent assignments in mathematics is through functions, and this is also the method preferred by Coq since it eases its use in proofs. Therefore, the state our program's transform can be though of as a function with the signature `Variable -> Value`.

As we explained in Chapter 2.1, the Ethereum Virtual Machine is a stack-based virtual machine with an added memory and storage models Since the Ethereum Virtual Machine is a low-level virtual machine, it works on the byte level, usually by grouping them in sequences of 32 called words. These words cover all possible values our variables can take.

```
Definition EVMWord := word EVMWordSize.
(* EVMWordSize = BytesInEVMWord * EVMByteSize
               =          32 * 8
               =           256              *)
```

In a smart contract the values of variables live either in memory or in storage, while the stack is instead reserved for temporary values. Therefore it is safe to assume that the values we are interested in will be stored in memory or storage. For the purposes of the model, whether a variable is stored in memory or storage is not relevant, and so we make no such distinction. Since in the execution of a program there is a finite number of variables, each can be represented by a natural number. We can then represent the status of all variables in our program to be a function from `nat` to `EVMWord`, which is precisely how they are defined in `constraints.v`.

```
Definition assignment : Type := nat -> EVMWord.
```

EVM programs can then be represented as functions which take an initial state of type `assignment`, transforms it, and returns a final state, also of type `assignment`. Since a program is just a function of type `assignment -> assignment` we can define semantic equality through the principle of functional extensionality, which says that if for all possible inputs two functions output the same values then the functions must be the same.

```
Axiom functional_extensionality : forall {X Y: Type}
                                          {f g : X → Y},
  (forall (x:X), f x = g x) → f = g.
```

Functional extensionality is not native to Coq, since it cannot be proven directly. Therefore, if we wanted to use it we would need to add it as an axiom. Fortunately, adding this axiom is known to keep Coq's logic consistent, which means that we cannot prove a proposition simultaneously to be true and false. Since it is harmless, Coq provides it as a definition in its standard library, under `Coq.Logic.FunctionalExtensionality`, although not in the form described before. Coq's functional extensionality axiom is tweaked to also work with functions which return dependent types. Our `functional_extensionality` proposition is then derived from this axiom.

```
Axiom functional_extensionality_dep : forall {A} {B : A -> Type},
  forall (f g : forall x : A, B x),
  (forall x, f x = g x) -> f = g.
```

Unfortunately, proceeding with this approach is unfeasible since testing all possible assignments for the complete program is too complex. Therefore, the approach followed by `FORVES` is different. Instead, it records the effects of the execution of both programs on a generic model of the state and simplifies them to a canonical state to see if they are equivalent, by applying simplification rules. Furthermore, `FORVES` does not test the transformation performed by the whole program, but only of jump-free sequences of instructions. This had the added benefit of not having to worry about conditionals with the added cost that we can no longer make any assumptions about the stack. However this is easily fixed by extending our `assignment` definition to also include variables representing elements in the stack, not only offsets in memory.

Furthermore, we may also not need to consider all states, since not all states are reachable at a particular point in the code. For instance, if a piece of code is only reachable through a conditional statement which checks for an invariant P, we can assume that the invariant holds for the final assignment. Consider the following snippets of code.

```
if(x > y){
    z = x-y;
}

if(x > y){
    z = abs(x-y);
}
```

The result of executing the section inside the `if` statement in the first snippet is setting the variable `z` to `x-y`, while in the second snippet it is set to `abs(x-y)`. Both of these effects are not equivalent if we consider an initial state of `{x:0, y:1}` since then $x - y = -1 \neq 1 = |-1|$. However, the condition in the `if` statement discards these types of states. We can represent this by making the condition in the `if` statement a requirement for the model to follow.

Therefore, for some optimizations we may want to ask for some special condition to be applied. For example, in the previous example we may consider simplifying `abs(z)` to `z` if we are able to show that `z > 0`. Sometimes, these conditions are not explicitly stated in the context, but can instead be derived from it. Consider an optimization `add_zero`, which says that if we have `x0 + x1` and `x0` is zero then this is equivalent to just having `x1`. This optimization can only be applied if we have the condition `x0 = 0`. Now, consider the following snippet.

```
if (x > 0) f();
else if (x < 0) g();
else {
    y = y + x;
}
```

When we reach the `else` branch the contextual information we have recorded is that $x \leq 0$ and $x \geq 0$, but not $x = 0$. However, we know that having $x \leq 0$ and $x \geq 0$ implies $x = 0$. Therefore, to apply the optimization `add_zero` we need to be able to show that the current contextual information we have implies the conditions we need, not that it is present explicitly.

3.2. Representation of constraints

How do we represent this in Coq? For starters, we need to define what type of contextual information are we working with. Since the Ethereum Virtual Machine works at the byte level, all its comparisons are arithmetic, which means the only conditions we are interested in are equalities and inequalities.

```
Inductive constraint : Type :=
  | C_LT (l r : cliteral) (* l <  r *)
  | C_EQ (l r : cliteral) (* l == r *)
  | C_LE (l r : cliteral).(* l <= r *)
```

We call `cliterals` to the elements we compare. In general, we are interested in comparisons between variables, constants and variables offset by a constant. These are what our `cliterals` represent.

```
Inductive cliteral : Type :=
  | C_VAR (n : nat) (* x *)
  | C_VAL (n : N) (* c *)
  | C_VAR_DELTA (n: nat)(delta : N).(* x + c *)
```

Armed with these definitions we can start to represent what we mean by contextual information. This information is obtained from the optimizer when it is performing the static analysis of the program, in particular from the branches in the code. After each conditional jump, if the jump was taken we can add the condition tested to the context, and if it was not taken we add the negation of the condition. If we have multiple nested conditions, these can be combined logically by using an `and` connective. It could also happen that one same region of the code is reachable from different jump instructions, which themselves carry different contextual information. This is the case, for example, with function calls. In this case we can combine the contextual information through an `or` connective.

To ease the representation of contextual information, we represent these combined conditions in disjunctive normal form. In code, this is modeled as a list of lists, where the inner lists represent conjunctions and the outer list represents one big disjunction.

```
Notation conjunction := (list constraint).
Notation disjuntion := (list conjunction).
Definition constraints : Type := disjuntion.
```

3.3. Implication between constraints

The type `constraints` represents the context that is available at one point in the code. What we want to be able to do is check if some set of constraints imply a different one. But in order to define this we first need to define what it means that a set of constraints imply another.

In logic, implication is usually expressed by the symbol $\rightarrow$, although the symbol $\subset$ is sometimes also used for the same purpose. This last symbol sheds more light into how we might want to define implication in this case.

For any state, a condition `c` may or may not hold. We say that a model satisfies a single constraint if after substituting the variables by the values assigned to them by the model the condition holds.

```
Definition satisfies_single_constraint
    (model: assignment) (c: constraint) : bool :=
  let get_value := cliteral_to_nat model in
  match c with
  | C_EQ l r => (get_value l =? get_value r)%N
  | C_LT l r => (get_value l <? get_value r)%N
  | C_LE l r => (get_value l <=? get_value r)%N
  end.
```

We can extend this definition to conjunctions of constraints by requiring that the model satisfies all constraints at the same time.

```
Definition satisfies_conjunction
    (model: assignment) (conj: conjunction): bool :=
  forallb (satisfies_single_constraint model) conj.
```

Finally, we extend the definition to also hold for generic combinations of constraints in disjunctive normal form by specifying that at least one conjunction must hold for the whole formula to be true.

```
Definition satisfies_constraints
    (model: assignment) (cs: constraints): bool :=
  forallb (satisfies_conjunction model) cs.
```

Now, we can take inspiration from the $\subset$ notation of implication to define implication of constraints. We say that a set of constraints in disjunctive normal form `cs` implies a constraints `c` if, for every model which satisfies `cs`, it also satisfies `c`. That is, if $State_{\mathtt{cs}}$ is the set of all states which satisfy cs and $State_{\mathtt{c}}$ is the set of all states which satisfy c, then $State_{\mathtt{cs}} \subset State_{\mathtt{c}}$. In Coq, we would write

```
Definition imply(cs: constraints)(c: constraint) :=
  forall (m: assignment),
    satisfies_constraints m cs = true ->
    satisfies_single_constraint m c = true.
```

A similar definition follows for conjunctions instead of terms in DNF.

```
Definition conj_imply(cs: conjunction)(c: constraint) :=
  forall (m: assignment),
    satisfies_conjunction m cs = true ->
    satisfies_single_constraint m c = true.
```

3.4. Implication checker

We can finally define what we mean by an implication checker. In the end, an implication checker is just a function which takes some constraints as contextual information, which we will call the hypothesis, and a constraint we will call the

thesis, and returns the boolean true if it is able to show that the hypothesis imply the thesis.

In a different programming language we would implement implication checkers as functions with the signature `constaints -> constraint -> bool`, but in Coq we can go further and encode the implication checker's invariant as part of the type.

```
Record imp_checker: Type :=
  { imp_checker_fun: constraints -> constraint -> bool
  ; imp_checker_snd: forall (cs: constraints) (c: constraint),
      imp_checker_fun cs c = true -> imply cs c
  }.
```

An implication checker is not just the function which checks for the implication, but also the proof of that if the checking function says the implication is true, then it is a fact that the hypothesis imply the thesis.

To implement the implication checker, we need to handle the constraints in disjunctive normal form. However, we observe that in general we can consider the different conjunctions of constraints independently. The intuition behind it is that, if we need to prove that $A \vee B \rightarrow C$, then we need to prove $A \rightarrow C$ and $B \rightarrow C$ separately. This can actually be easily proven by using the identities $A \rightarrow B \equiv \neg A \vee B$ and the distributive property of $\wedge$ and $\vee$.

$$
\begin{aligned}
A \vee B \rightarrow C &\equiv \neg(A \vee B) \vee C \\
&\equiv \neg A \wedge \neg B \vee C \\
&\equiv (\neg A \vee C) \wedge (\neg B \vee C) \\
&\equiv (A \rightarrow C) \wedge (B \rightarrow C)
\end{aligned}
$$

In Coq we represent this by defining a different type of implication checker, one that only works with conjunctions, and showing that a conjunction implication checker can be used as a regular implication checker and vice versa.

```
Record conj_imp_checker: Type :=
  { conj_imp_checker_fun: conjunction -> constraint -> bool
  ; conj_imp_checker_snd: forall (cs: conjunction) (c: constraint),
      conj_imp_checker_fun cs c = true -> conj_imply cs c
  }.
```

That a `imp_checker` can work as a `conj_imp_checker` is obvious and not really useful, since a conjunction is trivially in disjunctive normal form, where there are no disjunctions. The interesting side of this equivalence is the converse, showing that given a `conj_imp_checker` we can create an `imp_checker`. We do this by defining the `mk_imp_checker` function, of signature `conj_imp_checker -> imp_checker`.

```
Program Definition mk_imp_checker
    (checker: conj_imp_checker): imp_checker := {|
```

```
  imp_checker_fun (cs : constraints) c :=
    match cs with
    | [] => false
    | _ =>
        forallb (fun conj => conj_imp_checker_fun checker conj c) cs
    end
|}.
```

Since we included the implication checkers invariant inside its type, Coq will not let us finish the definition of this function until we have proven that this function preserves the invariant. That is why we need to use the `Program` prefix to this definition, to reassure Coq that we know that the definition is not finished with just the function, and that we will follow with the proof of the invariant.

As we saw before, given a conjunction implication checker we can define a more general implication checker by running the conjunction implication checker on each conjunction in the hypothesis and making sure they all hold true. We now need to show that, given that the conjunction implication checker is sound, the implication checker derived from it is sound as well. Let's look at the proof of this.

We use the `Next Obligation` command to enter in proof mode after the `Program` definition we used before.

```
Next Obligation.
  (* First we unfold the definition of imply and introduce the model
     in its definition *)
  unfold imply; intros model.
  (* Then we obtain the conjunction checker's checking function and
     soundness proof from its constructor *)
  destruct checker as [checker checker_snd].
  (* Next Obligation automatically introduces as many terms as
     possible. In particular, the goal it asks to prove is whether an
     assignment called model satisfies the constraint c given that:
       H : the checker function we have derived says that cs imply c
     We will rename H to full_checker__cs_imp_c. Full checker is the
     checking function we have derived from the conjunction
     implication checker's *)
  rename H into full_checker__cs_imp_c.
  (* We proceed over induction on c. The base case is not reachable,
     so we discriminate. We are only left with the inductive case. *)
  induction cs as [|c' cs' IHcs']; try discriminate.
  simpl in *.
  (* Since we are left in the case where cs = c' :: cs' and the full
     checker said that cs imply c, by definition that means that
     c' imply c and cs' imply c. Notice that:
        cs': list list constraint (DNF)
         c': list constraint (conjunction)
         c : constraint                             *)
  apply Bool.andb_true_iff in full_checker__cs_imp_c
```

```
    as [checker__c'_imp_c checker__cs'_imp_c].
  (* Moreso, to show that the disjuntion c' \/ cs'' implies c we
     need to that the model satisfies c both when it satisfies c'
     and it satisfies cs'. *)
  intros h; apply Bool.orb_true_iff in h as [c'_sat | cs'_sat].
  - (* If the model satisfies c' then since the conjugation checker
       said that c' implies c, we have our result.*)
    exact (checker_snd _ _ checker__c'_imp_c model c'_sat).
  - (* Otherwise, we apply the induction hypothesis on cs' *)
    unfold is_model in cs'_sat.
    (* We can assume that cs' is nonempty, otherwise the proof would
       have concluded earlier *)
    destruct cs' as [|c'' cs'']; try discriminate.
    exact (IHcs' checker__cs'_imp_c cs'_sat).
Qed.
```

Thanks to this function, which we have now proven that correctly yields implication checkers from conjunction implication checkers, serves to ease the development of implication checkers. To define a conjunction implication checker we no longer need to deal with DNF formulas, just with conjunctions. We can later derive the full implication checker from the conjunction one.

Before we conclude this chapter, we must highlight an important fact. We have only required our implication checkers to be sound, but we have not said anything about completeness. This means that if an implication checker returns false we cannot assume that the implication is in fact false. Proving completeness is quite a hard endeavour, and was therefore considered out of scope for this project.

This means that there is some type of partial order between implication checkers, depending on if how close to completeness they are. We can implement some trivial implication checkers which are pretty far away from completeness.

One such example would be the implication checker that always returns false. But a more interesting example is the implication checker which just checks whether the thesis is in the hypothesis.

```
Program Definition inclusion_conj_imp_checker: conj_imp_checker := {|
  conj_imp_checker_fun := fun cs c => existsb (eqc c) cs
|}.
(* Proof left as an exercise to the reader *)
Definition inclusion_imp_checker :=
    mk_imp_checker inclusion_conj_imp_checker.
```

While we have defined our implication checkers to be best-effort, we of course would like to have the best implication checker we could possible have. That is the purpose of the next chapter.

Chapter 4

Implication checker from transitive closure

Having defined what an implication checker is, we are now left with the task of developing the most advanced implication checker we can muster. Originally I decided to approach this with the idea of modelling constraints as a graph, where our `cliterals` are the nodes of the graph and the constraints its edges. With this representation we would transform the problem of testing whether an constraint is implied into finding a path in a graph, for which many solutions are already known.

Unfortunately, this original approach had a problem, and it had to do with allowing our `cliterals` to include variables with an offset. By allowing x and $x + d$ for $d \in \mathbb{N}$ to be represented by different nodes, we no longer have a cheap way of expressing that if $x + d \geq y + d'$ then $x + d + k \geq y + d' + k$ for every $k \in \mathbb{Z}$. The graph idea may be salvaged, but our definition of `cliteral` could not be its nodes.

While investigating the prior work we came across [10], where the author describes a $O(n^3)$ algorithm to compute the tightened transitive closure of a set of octagonal constraints.

4.1. Tightened transitive closure of octagonal constraints

Octagonal constraints are constraints of the form $\pm x - \pm y \leq d$ or $\pm x \leq d$ where $x, y, d \in \mathbb{Z}$. We say that constraints of the form $\pm x - \pm y \leq d$ are addition constraints and of the form $\pm x \leq d$ are bound constraints.

Our constraints can be translated to octagonal constraints through the following transformations. First, we exchange every `C_EQ` constraint with two `C_LE` constraints and every `LT` constraints with a `LE` constraint by taking advantage of the fact that since these are integer constraints, if $x < y$ then $x \leq y - 1$. Afterwards, we get rid of all inequalities of the form $d \leq d'$ where d ad d' are constant values and not variables. After all, these inequalities do not reveal any information, since they are always true if the constraints are satisfiable, and we assume our constraints are indeed satisfiable. Finally, we transform all `cliterals` of the `C_VAR` constructor to a `C_VAR_DELTA` constructor, where the `delta` field is left as 0. From there, if our constraints are not already in the form of an octagonal constraint, it is enough to

multiple both sides by -1 or move around terms from one side of the inequality to the other until they do.

Given an octagonal addition constraint of the form $ax + by \leq d$ where $a, b \in \{-1, 1\}$, we say that $ax + by \leq d$ *trivially implies* $ax + by \leq d'$ if $d \leq d'$. We say that a set of constraints C' is the transitive closure of C if every constraint implied by C is trivially implied by a constraint in C'.

Therefore, if we had an algorithm to compute the transitive closure of a set of constraints C we could implement a conjunction implication checker testing whether the thesis is trivially implied by any of the constraints in the transitive closure of the hypothesis. This algorithm is our final goal.

In [10] another simpler algorithm is mentioned to compute the transitive closure of a set of octagonal constraints, first described in [12]. This algorithm relies on iteratively adding to an already transitively close set new additive constraints, along with all the constraints needed to keep the set transitively closed.

In particular, if A is a set of addition constraints and B is a set of bound constraints, where $A \cup B$ is transitively closed, then $A \cup B \cup \{ax + by \leq d\} \cup A' \cup B'$ is also transitively closed, where A' is formed by the addition constraints

A1 $ez + by \leq d + d'$ if $-ax + ez \leq d'$, $z \neq y$

A2 $ax + ft \leq d + d'$ if $ft - by \leq d'$, $t \neq x$

A3 $ex + ft \leq d + d' + d''$ if $-ax + ex \leq d'$, $-by + ft \leq d''$ and t, z, x and yz are all different.

and B' is formed by the bound constraints

B1 $by \leq d + d'$ if $-ax \leq d'$

B2 $ax \leq d + d'$ if $-by \leq d'$

B3 $ez \leq d + d' + d''$ if $-ax \leq d'$ and $-by + ex \leq d''$ with $z \neq x$

B4 $ft \leq d + d' + d''$ if $-by \leq d'$ and $-ax + ft \leq d''$ with $t \neq y$

B5 $by \leq \lfloor \frac{d+d'}{2} \rfloor$ if $-ax + by \leq d'$

B6 $ax \leq \lfloor \frac{d+d'}{2} \rfloor$ if $-by + ax \leq d'$

B7 $ez \leq \lfloor \frac{d+d'+d''}{2} \rfloor$ if $-ax + ez \leq d'$ and $-by + ez \leq d''$

With this result one can derive an algorithm to obtain the transitive closure of any set of octogonal constraints. Given C_A and C_B sets of addition and bound constraints respectively, we start with $B = C_B$ and $A = \{ax + by \leq d + d' \ : \ \forall ax \leq d \in C_B, by \leq d' \in C_B\}$ and iteratively add the elements in C_A using the rules above, taking care to remove any constraint which may be trivially implied by another.

Based on these rules we implemented a simpler algorithm which is just as powerful but with the downside of requiring more iterations to derive the transitive closure. The reason is that these rules are useful if we want to prove the completeness of the transitive closure, but since we are only interested in the soundness of the

algorithm, we can notice that some of these rules are actually the result of applying one or more of the other rules.

In particular, we remove the rules:

- **A3** since it can be derived from successive uses of **A1** or **A2**.
- **B3** since it can be derived from successive uses of **B1** and **B2**.

$$-ax \leq d' \qquad ax + by \leq d \overset{\boxed{\textbf{B1}}}{\Longrightarrow} by \leq d + d'$$

$$bx \leq d + d' \qquad -by + ex \leq d'' \overset{\boxed{\textbf{B2}}}{\Longrightarrow} ez \leq d + d' + d''$$

- **B4** since it's the symmetric of **B3**.
- **B7** since it can be obtained by successively applying **A1** and **B6** or **A2** and **B5**.

$$-ax + ex \leq d' \qquad ax + by \leq d \overset{\boxed{\textbf{A1}}}{\Longrightarrow} ex + by \leq d + d'$$
$$ex + by \leq d + d' \quad -by + ex \leq d'' \overset{\boxed{\textbf{B6}}}{\Longrightarrow} ex \leq \lfloor \tfrac{d+d'+d''}{2} \rfloor$$

If we eliminate the requirement that after each new constraint added the set must be transitively closed, we can actually go without these rules. After adding a new constraint we would just have to apply the rules also with the constraint we derived at that step until no new constraints can be added. And since we no longer care if the intermediate set of constraints is transitively closed or not, we can drop the requirement of adding the constraints iteratively and simply, for our currently built set of constraints, test for every pair whether they can be combined and add the result if its not trivially implied by any of the other constraints already in the set.

Therefore, the final rules we have are:

R1 $ax + by \leq d \quad -by + ez \leq d' \quad \Longrightarrow ax + ez \leq d + d'$

R2 $ax + by \leq d \quad -ax + ez \leq d' \quad \Longrightarrow by + ez \leq d + d'$

R3 $ax + by \leq d \quad ez - ax \leq d' \quad \Longrightarrow by + ez \leq d + d'$

R4 $ax + by \leq d \quad ez - bx \leq d' \quad \Longrightarrow by + ez \leq d + d'$

R5 $ax + by \leq d \quad -ax \leq d' \quad \Longrightarrow by \leq d + d'$

R6 $ax + by \leq d \quad -by \leq d' \quad \Longrightarrow ax \leq d + d'$

R7 $ax \leq d \quad by \leq d' \quad \Longrightarrow ax + by \leq d + d'$

The algorithm, in pseudocode, would look like this

```
function TransitiveClosure(C: constraints) -> constraints
    repeat
        new_constraints := [
            combine(c1,c2) for c1 in C for c2 in C
        ]
        C = join(C, new_constraints)
    until C not changed
    return C
```

We have to pay attention to not include unnecessary constraints when adding new ones to the set, which is why a `join` function is explicitly used. Applying the rules is left for the `combine` function.

4.2. Implementation of the implication checker

Since the representation of the constraints is not exactly the same for our new algorithm, we have opted to include these definitions in its own file, aptly named `octagon.v`. Implementing an implication checker as defined before in terms of the algorithm developed in this chapter is left as future work[1]. First of all, we need to define in Coq what are our octagonal constraints. Since there are two types of octagonal constraints, additive and bound constraints, we represent this with two different constructors.

```
Inductive Constraint :=
 | AddConstr (l r: term)(d: Z)
 | BndConstr (t: term)(d: Z).
```

In this definition, the terms represent the $\pm x$ elements in our constraints. To represent the $\pm$ in Coq we use the subset $\{1, -1\} \subset \mathbb{Z}$.

```
Definition pmUnit := { z : Z | z = 1 \/ z = - 1 }.
Record term := {a: pmUnit; x: nat}.
```

Variables in this representation are still identified with natural numbers, but since they can now take any integer values we need to also change our previous `assignment` type definition, which we call now the type of `models`.

```
Definition model : Type := nat -> Z.
```

Therefore, we say that a model `m` satisfies a single constraint `m` if the following function returns true.

```
Definition satisfies_single_constraint
    (m: model) (c: Constraint): bool  :=
  match c with
  | AddConstr l r d => term_value m l + term_value m r <=? d
  | BndConstr t d => term_value m t <=? d
  end.
```

[1]See Section 5.1 for more information.

term_value is a function which retrieves the value of the variable x according to the model m and multiplies it by a to get the value of the term under the model.

```
Definition term_value(m: model)(t: term): Z :=
    proj1_sig t.(a) * (m t.(x)).
```

In this new file we have also ported over a more general definition of implication of constraints, which works over conjunctions. Since we showed in Chapter 3 that conjunction implication checkers are equivalent to implication checkers, in this file we drop the requirement of disjunctions in the constraints and simply treat our hypothesis as conjunctions of constraints.

We also include a more general version of implication which works with a conjunction of thesis constraints, and add some notation to help us work with it more comfortably.

```
Definition implication(C: list Constraint)(C': list Constraint) :=
  forall m,
    satisfies_constraints m C = true ->
    satisfies_constraints m C' = true.
Infix "==>>" := implication (at level 96, right associativity).
```

In this definition, satisfies_constraints is implemented as a forallb over the satisfies_single_constraint function. Its definition reads as "for every model which satisfies C, it also satisfies C'".

The usual facts about implication are also true for ==>>, most notably.

- Reflexivity

```
Theorem implication_refl(C: list Constraint):
  (C ==>> C).
```

- Transitivity

```
Theorem implication_trans(C C' C'': list Constraint):
  (C ==>> C') -> (C' ==>> C'') -> (C ==>> C'').
```

With these new definitions we proceed to define the functions which will help us compute the transitive closure, or at least given a list of constraints C help bring it closer to its transitive closure.

Since our algorithm consists of performing successively a given number of iterations over the list we define an iterate which we will call consecutively to iteratively bring closer our constraints C to its transitive closure.

```
Definition iterate(C: list Constraint) : list Constraint :=
  let C' := new_constraints C in
  let C'' := flatten C' in
  join C C''.
```

If we de-sugar the definition, we will see that `iterate` is in fact the composition of three different functions:

- `new_constraints`
- `flatten`
- `join C`

`new_constraints` itself is just a function which takes every pair of constraints in `C` and collects all the constraints obtained from their combination. These combinations come from the rules in Harvey's and Stuckey's algorithm[12], but simplified to make our life easier. The rules themselves are implemented in the `combine` function.

```
Definition new_constraints(C: list Constraint): list Constraint :=
  flat_map (fun c =>
    flat_map (fun c' =>
        opt_to_list(combine c c')
    ) C
  ) C.

Definition combine(c c': Constraint): option Constraint :=
  match c, c' with
  | AddConstr l r d, AddConstr l' r' d' =>
      if      l =? (op l') then Some (AddConstr r r' (d + d'))
      else if l =? (op r') then Some (AddConstr r l' (d + d'))
      else if r =? (op l') then Some (AddConstr l r' (d + d'))
      else if r =? (op r') then Some (AddConstr l l' (d + d'))
      else None
  | AddConstr l r d, BndConstr t' d' =>
      if      l =? (op t') then Some (BndConstr r (d + d'))
      else if r =? (op t') then Some (BndConstr l (d + d'))
      else None
  | BndConstr t d, AddConstr l' r' d' =>
      if      l' =? (op t) then Some (BndConstr r' (d + d'))
      else if r' =? (op t) then Some (BndConstr l' (d + d'))
      else None
  | BndConstr t d, BndConstr t' d' => Some (AddConstr t t' (d + d'))
  end.
```

The `flatten` function is just a map over the function `normalize_constraints`, which performs a very simple transformation, those of the constraints of the form $x + x \leq d$, which turn into $x \leq \lfloor \frac{d}{2} \rfloor$.

```
Definition normalize_constraint(c: Constraint): Constraint :=
   match c with
   | AddConstr l r d =>
```

```
      if l =? r
      then BndConstr l (d / 2)
      else c
  | c => c
  end.
```

In fact, this is an important transformation, referred to as *tightening* in the literature [10]. Since we know that our variables hold integer values, if we have a constraint of the form $x + y \leq d + \epsilon$ where ϵ is a real number in the interval $(0, 1)$ and $x, y, d \in \mathbb{Z}$ then we know that $x + y \leq d$, since that is its closest integer value to $d + \epsilon$. Sometimes this can allow us to improve upon our original constraints. Consider, for example, that we had the following constraints.

$$x + y \leq 10 \tag{4.1}$$
$$x - y \leq 3 \tag{4.2}$$
$$y \leq 3 \tag{4.3}$$

Then, combining 4.1 and 4.2 we can deduce that $x + x \leq 13$, which we can tighten into the bound constraint $x \leq 6$. Then, combining this with 4.3 we obtain the addition constraint $x + y \leq 9$, which is an improvement over 4.1.

Finally the `join` is potentially one of the most complicated functions in this file. Its purpose is to merge two lists of constraints into one, but getting rid of those constraints which are duplicated or no longer needed.

```
Definition joined(c': Constraint)(cs: list Constraint) :=
  c' :: filter (fun c => negb (trivial_impl c' c)) cs.

Definition join(C C': list Constraint): list Constraint :=
  fold_left (fun cs c' =>
    if forallb (fun c => negb (trivial_impl c c')) cs
    then joined c' cs
    else cs
  ) C' C.
```

The property we want to prove for `iterate` is the following.

```
Theorem iterate_implication(C T: list Constraint):
  (C ==>> iterate C).
```

This is but a corollary of the following theorems.

```
Theorem flatten_implication(C T: list Constraint):
  (C ==>> T) -> (C ==>> flatten T).
Theorem join_implication(C T: list Constraint):
  (C ==>> T) -> (C ==>> join C T).
Theorem new_constraints_implication(C T: list Constraint):
  (C ==>> T) -> (C ==>> new_constraints T).
```

Once these are proven, the proof of iterate_implication is as simple as performing successive applications of these theorems.

```
Theorem iterate_implication(C: list Constraint):
  (C ==>> iterate C).
Proof.
  apply join_implication.
  apply flatten_implication.
  apply new_constraints_implication.
  apply implication_refl.
Qed.
```

The proofs for these theorems can be found in the file octagon.v in FORVES2's GitHub repository.

With this iterate function defined and the iterate_implication theorem proven, the only piece left to construct our implication checker is a function which would allow us to apply the iterate function over the original constraints a given number of times.

```
Fixpoint church_numeral{A: Type}(n: nat)(f: A -> A)(x: A):=
  match n with
  | O => x
  | S m => f (church_numeral m f x)
  end.
```

We call this function church_numeral since it actually implements a translation between Peano-encoded naturals (the nat type in Coq) and Church-encoded naturals, where the number n corresponds to the function which takes another function f and a value x and applies the function over the value n times.

With these definitions it is now trivial to construct an implication checker. The only decision left to make is how many times we want to iterate. A good choice ends up being twice the number of our hypothesis constraints. This is because when deriving the rules of our algorithm (implemented in combine) we took care in ensuring that any rule from Harvey's and Stuckey's algorithm[12] could be derived by applying two of our own, and their algorithm is proven to be complete. Even if this completeness is not proven in Coq, it ends up being the optimal choice.

```
Program Definition conj_trans_closure_checker
    (n: nat) : conj_imp_checker := {|
  conj_imp_checker_fun cs c :=
    let trans_closure := church_numeral n iterate cs in
    existsb (fun c' => trivial_impl c' c) trans_closure
|}.
Next Obligation.
(* The proof proceeds by induction. We informally
   prove it for the case n=1, assuming n=0 as proven.
   Let  c' be a constraint in the transitive closure
```

```
   such that [c'] ==>> [c]. Then, we have:
            cs ==>> iterate cs    from iterate_implication
    iterate cs ==>> [c']          from imply_refl
          [c'] ==>> [c]           from our hypothesis
   Through transitivity, we conclude that cs ==>> [c]. *)
Qed.
Program Definition trans_closure_checker(n: nat) : imp_checker
    := mk_imp_checker (conj_trans_closure_checker n).
```

Take note that the `mk_imp_checker` used in this module is not the one we defined in Chapter 3, since we are using a different definition to represent the state and constraints. Most notably, in this section we have used integers while we previously restricted ourselves to natural numbers. An explanation on how to bridge this implementation with the definitions in Chapter 3 will be described in Section 5.1.

Chapter 5

Conclusions and Future Work

In Chapter 4 we implemented an algorithm which, when given a list of octagonal hypothesis constraints and an octagonal thesis constraint, will determine whether the thesis constraint can be obtained from the hypothesis. If this algorithm determines that it is possible, we have also formally verified that the implication holds true, and can therefore be relied on by different parts of the project.

We can now employ our algorithm to solve the problem introduced in Chapter 2. The Listings 2.1 and 2.2 were shown to both leave the stack in the same configuration, but they could not be considered semantically equivalent unless some conditions were met. In particular, these conditions were:

$$x_4 = 0 \tag{5.1}$$
$$x_0 \geq x_2 + 32 \tag{5.2}$$

In Section 2.1.2 we proved that 5.1 and 5.2 could be derived under the following constraints.

$$\begin{aligned} &x_4 = x_5 \\ &x_5 = 0 \\ &x_0 \geq x_2 + 128 \end{aligned}$$

Now, however, we are able to use the implication checker implemented in Chapter 4 to test this. First, we need to transform our hypothesis constraints into octagonal constraints. To do so, we follow the instructions described in Chapter 3.

$$\begin{aligned} x_4 = x_5 &\Longrightarrow x_4 -_5 \leq 0 \ \wedge \ x_5 - x_4 \leq 0 \\ x_5 = 0 &\Longrightarrow x_5 \leq 0 \ \wedge \ -x_5 \leq 0 \\ x_0 \geq x_2 + 128 &\Longrightarrow x_2 - x_0 \leq -128 \end{aligned}$$

To ease the definition of multiple constraints, we make use of the following convenience functions.

```
Program Definition mkadd_pp(x: nat)(y: nat)(d: Z) :=
  AddConstr (Build_term 1 x) (Build_term 1 y) d.
Program Definition mkadd_pn(x: nat)(y: nat)(d: Z) :=
  AddConstr (Build_term 1 x) (Build_term (-1) y) d.
Program Definition mkadd_nn(x: nat)(y: nat)(d: Z) :=
  AddConstr (Build_term (-1) x) (Build_term (-1) y) d.
Program Definition mkbnd_p(x: nat)(d: Z) :=
  BndConstr (Build_term 1 x) d.
Program Definition mkbnd_n(x: nat)(d: Z) :=
  BndConstr (Build_term (-1) x) d.
```

With these functions, we define the list of constraints which represent the context of our application.

```
Local Definition C :=
  (* Obtained from [x4 = x5] *)
  [ Octagon.mkadd_pn 4 5 0    (*  x4 - x5 <= 0 *)
  ; Octagon.mkadd_pn 5 4 0    (*  x5 - x4 <= 0 *)
  (* Obtained from [x5 = 0] *)
  ; Octagon.mkbnd_p 5 0       (*  x5 <= 0*)
  ; Octagon.mkbnd_n 5 0       (* -x5 <= 0*)
  (* Obtained from [x0 >= x2 + 128] *)
  ; Octagon.mkadd_pn 2 0 (-128) (*  x2 - x0 <= -128 *)
].
```

This transformation we made for our hypothesis also needs to be made for our thesis. In particular, 5.1 transforms into the constraints $x_4 \leq 0$ and $-x_4 \leq 0$, which means we will need to check that both of them hold to conclude that it holds for 5.1.

```
Local Definition checker: Constraint -> bool :=
    conj_imp_checker_fun (conj_trans_closure_checker (2 * length C)) C.

(* x4 = 0 *)
Compute checker (Octagon.mkbnd_p 4 0) && checker (Octagon.mkbnd_n 4 0).
(* x0 >= x2 + 32 *)
Compute checker (Octagon.mkadd_pn 2 0 (-32)).
```

```
checker is defined

= true
     : bool

= true
     : bool
```

5.1. Future work

The original objective of this project was to develop an implication checker in Coq and verify its soundness. The objective of the project has therefore been achieved, however there is work to be done before it can be used in FORVES2 directly. While the algorithm is implemented and proven to be correct, before it can be used we need to connect it to the definitions found in constraints.v, those defined in Chapter 3.

The first definition to bridge would be those representing state, Octagon.model and Constraints.assignment. Both represent functions which transform naturals to values, but Octagon.model gives integers (Z) while Constraints.assignment gives naturals (N). The first step would be to show that Octagon.model contains Constraints.assignment, and to provide an explicit injection.

```
Definition translate_model:
    Constraint.assigment -> Octagon.model.
```

Afterwards, a transform of Constraints.constraints to Octagon.Constraints using the rules outlined in Chapter 3 would need to be defined. We can assume this transformation's signature to be something similar to the following.

```
Definition translate_constraints:
    Constraint.conjunction -> list Octagon.Constraint.
```

Then, we would need to show that our translate_constraints function preserves the information contained in the constraints. That is, some model satisfies some constraints if and only if the translated model also satisfies the translated constraints.

```
Lemma translate_preserves_information(C: list Constraint.constraint) :
  forall (m: Constraint.assigment),
    let m_oct = translate_model m in
    let C_oct = translate_constraints C in
      Constraint.satisfies_conjunction m C = true <->
      Octagon.satisfies_constraints m_oct C_oct = true.
```

Armed with these results we could define a Constraint.conj_imp_checker which would work by translating the constraints from the Constraint module to the Octagon's one and asking whether these imply the provided thesis constraints using Octagon.conj_trans_closure_checker. Notice that, since we have not defined a translation between single constraints, our initial single thesis constraint may now correspond to multiple ones. This can be solved by calling the checking function as many times as thesis constraints need to be proven, and ensuring that all of them hold. Using the translate_preserves_information lemma we would then show that the soundness of Octagon.conj_trans_closure_checker implies that of this checker.

Beyond the connection between the theories described in constraints.v and octagon.v, there is also the possibility of improving the runtime of the transitive

closure algorithm. For instance, [10] mentions an algorithm which uses multiple matrices to represent the collection of constraints. Beyond improving the algorithm's efficiency, this representation would simplify greatly the implementation of the `join` function.

One could also approach this from the perspective of generalizing our constraints container from the algorithm. A `Module Type` could be defined which implements the required operations and guarantees the necessary properties, and then the particular container could be developed independently.

Finally, as mentioned in Chapter 3, the `imp_checker` invariant only requires that if the checking function returns `true` then the implication holds. However, it would be useful to have the reciprocal: if the implication holds, then the checking function returns `true`. This could be used to develop a satisfiability checker.

Notice that our algorithm also constructs additive octagonal bounds of the form $x - x \leq d$. These constraints will hold as long as d is a non-negative integer, regardless of the assignment of x. In fact, [12] proves that deriving a constraint of the form $x - x \leq d$ for a negative d is sufficient and necessary to prove that the original constraints were unsatisfiable, that is, they are not satisfiable under any model.

If we had the completeness result, we could implement a satisfiability checker which checks whether the constraint $x{-}x \leq -1$ is implied by the original constraints, for any variable x. Then, if the checker is unable to say that the implication holds, through completeness we would be able to assert that the constraints are satisfiable.

While completeness is a nice property to have, it has never been a requirement of the `FORVES2` project. Thanks to the SMTCoq project[5] we can rely on external SMT solvers to prove that a set of restrictions is satisfiable and verify their results. That is why in this project we opted to focus in soundness over completeness. Nevertheless, it would be the most natural extension of the concepts discussed in this project.

Conclusiones y Trabajo Futuro

En el Capítulo 4 implementamos un algoritmo que, cuando recibe una lista de hipótesis de restricciones octogonales y otra restricción octogonal como tesis, determina si la tesis se puede obtener de las hipótesis. Si este algoritmo determina que es posible, también hemos verificado formalmente que la implicación siempre es correcta, y por lo tanto se puede utilizar en distintas partes del proyecto.

Ahora podemos emplear nuestro algoritmo para resolver el problema introducido en el Capítulo 2. En este vimos que los Fragmentos 2.1 y 2.2 dejan la pila de la EVM en el mismo estado tras su ejecución, pero que no se podían considerar semánticamente equivalentes a no ser que se cumplieran ciertas restricciones. En concreto, estas restricciones eran.

$$x_4 = 0 \tag{5.3}$$
$$x_0 \geq x_2 + 32 \tag{5.4}$$

En la Sección 2.1.2 probamos que 5.3 y 5.4 se podían derivar de las siguientes restricciones.

$$\begin{aligned} & x_4 = x_5 \\ & x_5 = 0 \\ & x_0 \geq x_2 + 128 \end{aligned}$$

Ahora, sin embargo, podemos usar el comprobador de implicaciones implementado en el Capítulo 4 para comprobarlo. Primero, tenemos que transformar nuestras hipótesis en restricciones octogonales. Para ello, seguimos las instrucciones descritas en el Capítulo 3.

$$\begin{aligned} x_4 = x_5 &\Longrightarrow x_4 - _5 \leq 0 \ \wedge \ x_5 - x_4 \leq 0 \\ x_5 = 0 &\Longrightarrow x_5 \leq 0 \ \wedge \ -x_5 \leq 0 \\ x_0 \geq x_2 + 128 &\Longrightarrow x_2 - x_0 \leq -128 \end{aligned}$$

Para facilitarnos la construcción de varias restricciones, usamos las siguientes funciones de conveniencia.

```
Program Definition mkadd_pp(x: nat)(y: nat)(d: Z) :=
  AddConstr (Build_term 1 x) (Build_term 1 y) d.
Program Definition mkadd_pn(x: nat)(y: nat)(d: Z) :=
  AddConstr (Build_term 1 x) (Build_term (-1) y) d.
Program Definition mkadd_nn(x: nat)(y: nat)(d: Z) :=
  AddConstr (Build_term (-1) x) (Build_term (-1) y) d.
Program Definition mkbnd_p(x: nat)(d: Z) :=
  BndConstr (Build_term 1 x) d.
Program Definition mkbnd_n(x: nat)(d: Z) :=
  BndConstr (Build_term (-1) x) d.
```

Con estas funciones, podemos definir la lista de restricciones que representan la información del contexto de nuestra aplicación.

```
Local Definition C :=
  (* Obtained from [x4 = x5] *)
  [ Octagon.mkadd_pn 4 5 0    (*  x4 - x5 <= 0 *)
  ; Octagon.mkadd_pn 5 4 0    (*  x5 - x4 <= 0 *)
  (* Obtained from [x5 = 0] *)
  ; Octagon.mkbnd_p 5 0       (*  x5 <= 0*)
  ; Octagon.mkbnd_n 5 0       (* -x5 <= 0*)
  (* Obtained from [x0 >= x2 + 128] *)
  ; Octagon.mkadd_pn 2 0 (-128) (*  x2 - x0 <= -128 *)
].
```

Esta transformación de nuestras hipótesis también necesita realizarse para nuestra tesis. En concreto, 5.3 se transforma en las restricciones $x_4 \leq 0$ y $-x_4 \leq 0$, por lo que tendremos que comprobar que ambas se cumplen para poder decir que 5.3 se cumple.

```
Local Definition checker: Constraint -> bool :=
    conj_imp_checker_fun (conj_trans_closure_checker (2 * length C)) C.

(* x4 = 0 *)
Compute checker (Octagon.mkbnd_p 4 0) && checker (Octagon.mkbnd_n 4 0).
(* x0 >= x2 + 32 *)
Compute checker (Octagon.mkadd_pn 2 0 (-32)).
```

```
checker is defined

= true
     : bool

= true
     : bool
```

5.2. Trabajo futuro

El objetivo original de este trabajo era el desarrollar el comprobador de implicaciones y verificar su corrección. Por lo tanto, el objetivo del trabajo se puede dar por concluido. Sin embargo, aún queda trabajo por hacer para que este algoritmo se pueda usar directamente en el proyecto `FORVES2`. Mientras que el algoritmo se ha verificado, no se puede usar con las definiciones descritas en el Capítulo 3, que se pueden encontrar en el archivo `constraints.v`. Por lo tanto, sería necesario conectar estas definiciones con las usadas en la teoría del Capítulo 4.

Los primeras definiciones a conectar serían las del estado: `Octagon.model` y `Constraints.assignment`. Ambas representan funciones que toman números naturales, pero `Octagon.model` devuelve enteros (`Z`) mientras `Constraints.assignment` devuelve naturales (`N`). El primer paso sería mostrar que `Octagon.model` contiene a `Constraints.assignment`, y dar una inyección explícita.

```
Definition translate_model:
    Constraint.assigment -> Octagon.model.
```

También sería necesario implementar la traducción de restricciones descrita en el Capítulo 3, de `Constraints.constraints` a `Octagon.Constraints` usando las reglas descritas en el Capítulo 3. Podemos asumir que el tipo de esta transformación sería similar al siguiente.

```
Definition translate_constraints:
    Constraint.conjunction -> list Octagon.Constraint.
```

Finalmente deberíamos demostrar que la función `translate_constraints` preserva la información contenida en las restricciones. Es decir, que un modelo satisface unas restricciones si y solamente si el modelo traducido también satisface la transformación de las restricciones.

```
Lemma translate_preserves_information(C: list Constraint.constraint) :
  forall (m: Constraint.assigment),
    let m_oct = translate_model m in
    let C_oct = translate_constraints C in
      Constraint.satisfies_conjunction m C = true <->
      Octagon.satisfies_constraints m_oct C_oct = true.
```

Con estos resultados podríamos definir un `Constraint.conj_imp_checker` que funcionaría traduciendo las restricciones del modulo `Constraint` a las de `Octagon` y preguntando si se pueden derivar las tesis transformadas de las hipótesis con `Octagon.conj_trans_closure_checker`. Cabe recalcar que como nuestras transformaciones están definidas para conjuntos de restricciones, una restricción al traducirse puede resultar en varias restricciones. En estos casos se comprobaría que cada una de las restricciones resultantes se cumple para verificar que se cumple la tesis. Usando el lema `translate_preserves_information` podríamos probar que la corrección de `Octagon.conj_trans_closure_checker` implica la de nuestro comprobador.

Más allá de la conexión entre las teorías descritas en `constraints.v` y `octagon.v`, también existe la posibilidad de mejorar el tiempo de ejecución del algoritmo de clausura transitiva. Por ejemplo, [10] menciona un algoritmo que utiliza matrices para representar una colección de restricciones. Además de ser una representación más eficiente, también simplificaría la implementación de la función `join`.

También se podría abordar la cuestión desde el punto de vista de desacoplar al algoritmo de la implementación del contenedor de restricciones, mediante la definición de un `Module Type` que especifica las operaciones necesarias del contenedor y sus invariantes. De esta manera se podría desarrollar la teoría del comprobador de implicaciones independientemente de la implementación del contenedor.

Por último, como se mencionó en el Capítulo 3, el invariante `imp_checker` solo especifica que cuando la función comprobadora devuelve `true` entonces la implicación se cumple. Sin embargo, sería útil tener el recíproco: si la implicación se cumple, entonces sabemos que la función comprobadora devuelve `true`. Esta propiedad podría usarse para desarrollar un comprobador de satisfabilidad, por ejemplo.

Hay que tener en cuenta que nuestro algoritmo construye restricciones octogonales aditivas de la forma $x - x \leq d$. Estas restricciones son ciertas o no dependiendo del valor de la constate d e independientemente del valor de la variable x. De hecho, [12] prueba que derivar una restricción de la forma $x - x \leq d$ con d negativo es condición suficiente y necesaria para concluir que las restricciones originales son insatisfacibles, es decir, no son satisfacibles para ningún modelo.

Si tuviéramos el resultado de completitud, podríamos implementar un comprobador de satisfabilidad comprobando si la restricción $x - x \leq -1$ se puede derivar de las restricciones originales, para cualquier variable x. De esta manera, si el comprobador es incapaz de decir que la implicación es cierta, por la completitud podemos deducir que las restricciones son satisfacibles.

Aunque la completitud es una propiedad conveniente, esta nunca ha sido un requisito en el proyecto `FORVES2`. Gracias al proyecto SMTCoq[5] podemos usar resolutores SMT externos para probar que un conjunto de restricciones es satisfacible y verificar sus resultados. Es por esto que en este trabajo optamos en concentrarnos en la correctitud antes que la completitud del algoritmo. Aun así, es cierto que esta sería la extensión más natural de los conceptos tratados en este trabajo.

Bibliography

[1] Iris Project — iris-project.org. `https://iris-project.org/`, ???? [Accessed 13-05-2024].

[2] RustBelt — plv.mpi-sws.org. `https://plv.mpi-sws.org/rustbelt/`, ???? [Accessed 13-05-2024].

[3] Annenkov, D., Nielsen, J. B. y Spitters, B. Concert: a smart contract certification framework in coq. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020.

[4] Chu, S., Wang, C., Weitz, K. y Cheung, A. Cosette: An automated prover for sql. En *CIDR*. www.cidrdb.org, 2017.

[5] Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A. y Barrett, C. SMTCoq: A plug-in for integrating SMT solvers into Coq. En *Proceedings of the* 29th *International Conference on Computer Aided Verification (CAV '17)* (editado por R. Majumdar y V. Kuncak), vol. 10426 de *Lecture Notes in Computer Science*, páginas 126–136. Springer, 2017. Heidelberg, Germany.

[6] Erbsen, A., Philipoom, J., Gross, J., Sloan, R. y Chlipala, A. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. En *2019 IEEE Symposium on Security and Privacy (SP)*, páginas 1202–1219. 2019.

[7] Leroy, X. A formally verified compiler back-end. *J. Autom. Reason.*, vol. 43(4), página 363–446, 2009. ISSN 0168-7433.

[8] Parker, C. What is the runtime/time complexity of coq's (dependent) type inference? Computer Science Stack Exchange, ???? URL:https://cs.stackexchange.com/q/147849 (version: 2021-12-21).

[9] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V. y Yorgey, B. *Logical Foundations*, vol. 1. Software Foundations, 5.3 edición, 2017.

[10] Revesz, P. Tightened transitive closure of integer addition constraints. 2009.

[11] Sozeau, M., Forster, Y., Lennon-Bertrand, M., Nielsen, J. B., Tabareau, N. y Winterhalter, T. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq, 2023. Working paper or preprint.

[12] W. Harvey, P. S. A unit two variable per inequality integer constraint solver for constraint logic programming. 1997.

[13] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, vol. 151, páginas 1–32, 2014.