# Diseño e implementación del compilador de Circom 1.0
# Design and implementation of the Circom 1.0 compiler


**Trabajo de Fin de Máster**
**Curso 2019–2020**

**Autor**
Hermenegildo García Navarro

**Director**
Albert Rubio Gimeno

**Máster en Métodos Formales**
**Facultad de Informática**
**Universidad Complutense de Madrid**

# Diseño e implementación del compilador de Circom 1.0

# Design and implementation of the Circom 1.0 compiler

**Trabajo de Fin de Máster en Métodos Formales**
**Departamento de Sistemas informáticos y computación**


**Autor**
**Hermenegildo García Navarro**

**Director**
**Albert Rubio Gimeno**


**Convocatoria:** *Junio 2020* **Calificación:** *9*


**Máster en Métodos Formales**
**Facultad de Informática**
**Universidad Complutense de Madrid**


**20 de julio de 2020**

# Autorización de difusión

El abajo firmante, matriculado en el Máster de métodos formales de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: Design and implementation of the Circom 1.0 compiler, realizado durante el curso académico 2019-2020 bajo la dirección de Albert Rubio Gimeno en el Departamento de Sistemas informáticos y computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Hermenegildo García Navarro

20 de julio de 2020

# Acknowledgements

First of all I would like to thank Albert Rubio, advisor of this thesis, for trusting me for this project. Designing and implementing the Circom 1.0 compiler has not been an easy task but, thanks to Albert, it has been also an exciting task where I have grown as a computer scientist and as an engineer.

Thanks to my mother, for instilling in me a work ethic that has helped me throughout this project. Thanks to my father, for passing to me his interests in computer science. I do not often have a chance to thank my father for all the discussions we have had over the years about computer science, thank you. Thanks to both of them, for being an incredible source of support and knowledge.

Thanks to Marina, for supporting me in the most difficult moments of this project and for making the good moments great. If Christopher McCandless was right and happiness is only real when shared, I would like to thank her for all these years of making my happiness real.

Finally, thanks to Jack Kerouac, for showing us the road.

# Abstract


## Design and implementation of the Circom 1.0 compiler

The central element of the Circom project is the Circom programming language. This programming language is used for designing arithmetic circuits, which will be used in the field of cryptography. Circom programs, like most computer programs, are a set of instructions that can be executed by a computer. In this case, the execution mimics the behaviour of some arithmetic circuit. Unlike most computer programs, Circom programs are also an executable specification that, when interpreted, will produce the set of constraints of the arithmetic circuit. The Circom programming language is unique because its expressiveness allows compressing in the same code the behaviour of an arithmetic circuit and its restrictions.

To take advantage of such a special programming language the Circom compiler is needed. Like Circom programs, the Circom compiler encapsulates two different behaviours. When given a Circom program the Circom compiler will translate the circuit into a target language so the circuit can be executed. The Circom compiler will also behave as an interpreter and treat the Circom program as an executable specification, by executing this specification the Circom compiler will produce the constraints of the given circuit.

The Circom 1.0 compiler is a new compiler for this programming language partly developed for this thesis. The part developed for this thesis is formed by the structure and semantics analyses of Circom programs and the constraint generation. These elements of the Circom 1.0 compiler are the ones that take most advantage of the knowledge obtained in the master on formal methods. This thesis will start by providing some context to the Circom project, its history and where it is used. Following this will be an introduction to the Circom programming language and an overview of the elements that form the Circom 1.0 compiler. Most of this thesis length will be used to explain the type analysis of Circom programs, the unknown-known analysis and the constraint generation. This thesis also includes a chapter of experiments, where the perfomance and scalability of the Circom 1.0 compiler is tested. To conclude this thesis some conclusions will be given and the future work will be exposed.



## Keywords

Compiler, Interpreter, Circom, Static analysis, Type analysis, Symbolic execution, Arithmetic circuits, Constraint generation.

# Resumen


## Diseño e implementación del compilador de Circom 1.0

Circom, como proyecto, tiene de elemento central el lenguaje de programación Circom. Este lenguaje de programación es usado para diseñar circuitos aritméticos, que serán usados en el campo de la criptografía. Al igual que los programas escritos en otros lenguajes, los programas escritos en Circom son un conjunto de instrucciones que pueden ser ejecutadas por un computador. La diferencia de los programas escritos en Circom es que estos también son una especificación ejecutable que, al ser interpretada, produce el conjunto de restricciones del circuito aritmético.

Para aprovechar todo el potencial de este lenguaje de programación se necesita el compilador de Circom. Al igual que los programas escritos en Circom, este compilador agrupa dos comportamientos distintos. Cuando este compilador recibe un programa escrito en Circom este será traducido a un lenguaje destino. Por otro lado, el compilador de Circom se comportará como un intérprete del programa recibido y producirá el conjunto de restricciones del circuito.

Para este trabajo de fin de máster se ha desarrollado parte de la versión 1 del compilador de Circom. Las partes elegidas para este trabajo de fin de máster son los análisis semánticos y estructurales de los circuitos y la generación de restricciones. Estos elementos son los que mejor representan como la versión 1 del compilador de Circom aprovecha el conocimiento adquirido en el máster de métodos formales para mejorar las versiones anteriores.

Esta memoria comenzará aportando un contexto al proyecto Circom, contando su historia y donde es usado. A continuación se dará una introducción al lenguaje de programación y un resumen de los elementos que forman parte de esta nueva versión del compilador. La mayor parte de esta memoria esta dedicada al análisis de tipos, al análisis unknown-known y a la generación de restricciones. Esta memoria también incluye un apartado de experimentos, donde se ha puesto a prueba la escalabilidad y eficiencia de esta nueva versión del compilador. Esta memoria terminará exponiendo las conclusiones obtenidas de este trabajo y los avances que se harán en el futuro.



## Palabras clave

Compilador, Intérprete, Circom, Análisis estático, Análisis de tipos, Ejecución simbólica, Circuitos aritméticos, Generación de restricciones.

# Contents

# List of figures

# List of tables

Chapter 1

# Introduction

The goal of this thesis is to design and implement the Circom 1.0 compiler. This chapter approaches Circom from a practical point of view. From this chapter the reader will find answers to what Circom programs are and what their use in the real world is. To conclude the introduction, an overview of the thesis will be given.

## 1.1. Arithmetic circuits

Just like electronic circuits, arithmetic circuits are formed by gates that receive some signals as inputs and, by performing some arithmetic operations over them, produce some signals as outputs. Circom programs describe arithmetic circuits by means of a set of contraints that define their behaviour. The following picture shows a NAND gate, which is a circuit.

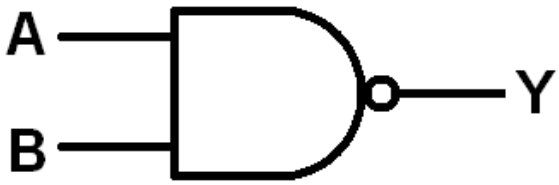



Figure 1.1: NAND gate, a simple circuit

This gate is also an arithmetic circuit where the result of performing $1 - AB$ will be the output. It has the following thruth table:

| A | B | C |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 1.1: NAND thruth table

Table 1.1 can be represented as the singleton set of constraints containing $Y = 1 - AB$. Just like electronic circuits, arithmetic circuits can get more complex by connecting different gates or performing complex computations over the inputs to produce the outputs.

When arithmetic circuits become more complex than the one seen in figure 1.1 designing them and computing its set of constraints gets harder. Given an arithmetic circuit, Circom programmers should be capable of codifying it as a Circom program that also contains the executable specification needed for computing its set of constraints.

## 1.2. Related languages

Circom is a DSL (domain-specific language) for writing zkSNARKs but is not the only one, in this section we will mention other DSLs for zkSNARKs. Snarky [8] is an OCaml front-end for writing R1CS SNARKs. Circom and Snarky programs are representations of the circuit that is being build. The difference between both of them is found in the programming paradigm used, Snarky has a functional nature and Circom is an imperative DSL based on C syntax.

A common aspect between Circom and Snarky is that both languages are low level with respect to circuit construction. This gives them the advantage of being more flexible in the level of precision for describing the circuit. There are other DSLs for writing zkSNARKs that try to abstract the programmer from the technicalities of building R1CS circuits. Two DSLs found in this category are ZoKrates [2] [21], with a Python like syntax and Zinc [10] [9], which follows Rust syntax.

## 1.3. A non-standard compiler

What compilers are can be simplified to them being computer programs that translate a program written in one language into a program written in another language; *gcc*, *rustc* and *solc* are examples of compilers. A compiler is not the same as an interpreter because the latter receives as input an executable specification and produces as output the result of executing the specification. Some languages such as *Perl* or *Python* are often implemented as interpreters. More information about compilers and interpreters can be found in [16].

When given a Circom program the Circom 1.0 compiler will run a series of algorithms to decide whether the input is a valid Circom program or not. One of the characteristics that makes the Circom 1.0 compiler special is that a Circom program is only valid if all the constraints are of the form $A * B - C = 0$ where $A$, $B$ and $C$ are linear combinations of the signals in the circuit. In coming chapters the reader will notice that the majority of the code in a Circom program is destined to telling the Circom 1.0 compiler how to build the set of constraints. Before translating the Circom program to a target language, the Circom 1.0 compiler needs to execute this code to obtain the set of constraints.

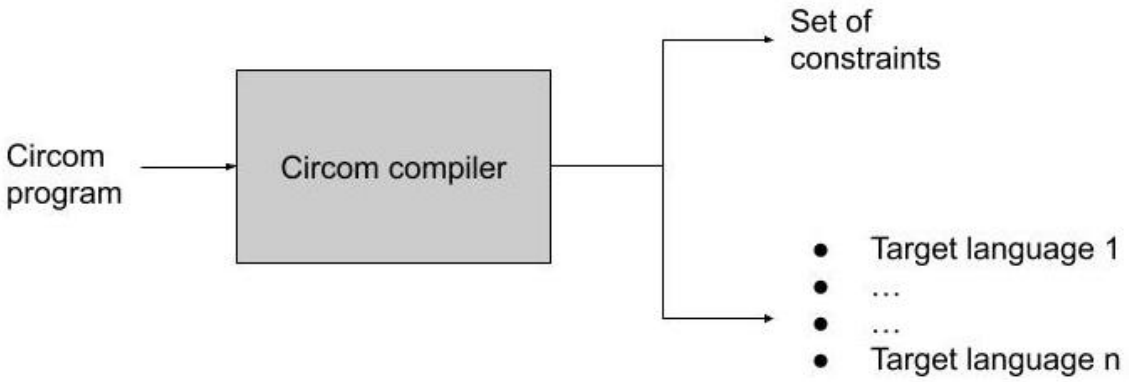



Figure 1.2: Overview of the Circom 1.0 compiler

As shown in figure 1.2 the ouptut of the Circom 1.0 compiler is formed by the arithmetic circuit translated to target languages so it can be executed and the set of constraints. This means that a Circom program is also an executable specification that the Circom 1.0 compiler will execute to get the set of constraints.

## 1.4. 0kims, the iden3 project and zero-knowledge proofs

The non-profit organization 0kims [1] develops open-source zero-knowledge solutions for identity management. Seeking this, 0kims creates iden3 [4], a set of tools for creating and managing self-sovereign identities on public blockchains [17]. It is, as a tool inside this project, where the Circom programming language borns.

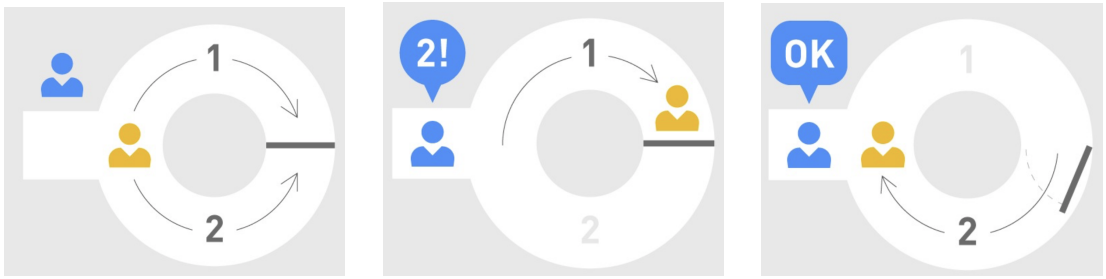
The first post about circom was on September of 2018. Till the Circom 1.0 compiler gets released the community is using the Circom 0.5 [3] which, like the Circom 0, is developed in Javascript. The Circom 1.0 compiler is a contribution to iden3 that covers the holes in the Circom programming language specification and gives an implementation of the compiler best suited for the increasing use of the language.

The rest of this brief subsection has the goal of giving to the reader a quick look at the world where Circom is used. As disclaimed at the start of the chapter, getting to understand the behavioral details of this world is quite difficult so this thesis will stay at a high level.

Zero-knowledge proofs [20] are the central axis of this world. Using them, a party (prover) can prove to another one (verifier) that they know a value $x$ using just the fact that they know the value. For newcomers to this concept (like myself) a great and easy example of a zero-knowledge proof can be found in [14]. The scenario where this example takes place can be seen in figure 1.3.

Bob and Alice know this cool cave shaped like a donut where walking through the paths does not take you back to the starting point. Instead it takes you to a misterious door that will open when the right words are said. One day Bob claims that some misterious man told him the words but if he were to tell them to someone terrible things would happen. At this point Alice, that has been studying criptography for a while, realizes that Bob could use a zero-knowledge proof to prove her that he knows the words without revealing them.

Alice will stay outside the cage and Bob inside it, once Bob arrives to the door Alice comes inside and shouts "one" or "two" to tell Bob the path she wants him to take for coming back. By repeating this experiment several times Alice can be sure that Bob can not be taking the right path every time out of luck so he must know the password.




(a) Bob (yellow) is deciding the path he will take

(b) Bob decides to take the path with label 1 and Alice shouts which path should he take for comming back

(c) Bob did take the right path

Figure 1.3: Alice (blue) and Bob (yellow) perfom a zero-knowledge proof

Zero-knowledge proofs are used to solve some problems surrounding the creation and managment of self sovereign identities on public blockchains. To produce a proof in an efficient way its common to use zk-snarks so they are short enough to publish to a blockchain.

A programming language for writting arithmetic circuits comes into play because most of the problems where a zero-knowledge proof is needed can be converted into them. Using Circom these circuits can be developed in a modular and reusable way.

## 1.5. The coming chapters

Designing and implementing the Circom 1.0 compiler has required a lot of work and many design choices had to be taken. Although I would like to talk about everything, this thesis will focus on the parts that make the Circom 1.0 compiler non-standard and are closely related with some of the courses taught at the master on formal methods.

After this introduction this thesis is going to dive into the Circom programming language. In this chapter the reader will obtain the knowledge needed to understand the input of the Circom 1.0 compiler and, hopefully, will arrive to the conclusion that Circom programs are not like other computer programs. The Circom programming language is still in an early stage so designing and implementing the Circom 1.0 compiler most of the time required thinking with my advisor, Albert Rubio, and the Iden3 project team about the language design (e.g, what should be allowed in the Circom programming language). This chapter is just a starting point for the thesis and the rest of the work focuses on what the Circom programming language is and what it is not.

This thesis puts the main focus on the analyses that are implemented in the Circom 1.0 compiler and can not be found in standard compilers. This thesis will end presenting the work done so the Circom 1.0 compiler generates the set of constraints defined in the given Circom program. Translating Circom programs into a target language is not covered in this thesis since during that process the Circom 1.0 compiler behaves like the standard compiler.

Chapter 2

# The Circom programming language

Circom allows the programmer to design circuits in a modular and efficient way. It also gives the possibility of writing inside the program an executable specification that will be used by the Circom 1.0 compiler to generate the constraint set of the circuit.

In this chapter the reader can find how the Circom programming language combines the executable specification of the constraints with the circuit design while also looking at some of the elements that can be found in a Circom program.

## 2.1. Designing circuits

Circuits are all about signals, through them they receive and produce information. For example, the circuit in figure 1.1 has three signals: two inputs and one output. Circom's syntax for signals is defined in the following way.

```
Signal ::= 'signal'Tag Type name Dim;
Type ::= 'input' | 'output' | ε
Tag ::= ':Binary' | ':FieldElement' | ε
Dim ::= '['expression']'Dim | ε
name ::= '_'*(A–Z + a–z + _ + 0–9)*
```

In the cases where Circom expressions are like the standard ones, this chapter will not get into all of them but the reader can get a vibe through the examples.

```
1   signal:Binary input a;
2   signal input b[12];
3   signal c[3][4];
```

Figure 2.1: Signal declaration examples

Using the Type production the programmer expresses whether the signal is an input or an output for the circuit. If the signal definition does not include a type then it is considered *intermediate* so it will be used only inside the circuit. Signals also can have as many dimensions as needed. Tag production generates what is called signal tags, they are related with the constraint generating behaviour of Circom program that will be covered later in this chapter.

The simplest way for passing values from one signal to another one is by using the operators –> and <–. During execution, a signal must not receive a value more than once.

For example $A$ –> $B$ means that the value of A goes to B.

The behaviour of a circuit is encapsulated inside the so called templates. In order to be reusable, they accept parameters that must be instatiated when the template is used. Using templates means creating what are called components and they represent concrete instances of a circuit. The relation between templates and components is similar to the one between classes and objects in object oriented programming. Templates and components are defined in the following way.

```
Template ::= template name(args) block
Component ::= component name Dim;
```

Components are initialized using the symbol '=' and their signals are accessed using dot notation, only the inputs and outputs declared in the template are accessible. The statements allowed inside a template are all the statements allowed in circom with the exception of template and function declarations, and they are the following ones.

```
if-then-else: if(expression) block else block
while loops: while(expression) block
for loops: for(statement;expression;statement) block
blocks: '{' statements '}'
Component declarations
Signal declarations
Variable declarations
Component assignments
Signal assignments
Variable assignments
```

Variables and functions will be mentioned later in this chapter. As can be seen the modular design of circuits is encouraged in Circom by allowing to declare components inside templates. This way the programmer is able to design circuits by using smaller ones. The only component declared outside a template is the *main* component and it represents the entry point to the circuit defined by the Circom program. When using components as part of templates the programmer should know that, like signals, components can be initialized just once.

Let us start looking at some examples. The following Circom program is a circuit that takes two inputs, adds them and outputs the result through a signal.

```
1  template Add() {
2      signal input val_0;
3      signal input val_1;
4      signal output ret;
5      ret <== val_0 + val_1
6  }
7  component main = Add();
```

Figure 2.2: Adder with two inputs

To generalize this circuit for $n$ inputs, let us look at two new elements of the Circom programming language: variables and functions. Variables can hold arithmetic expressions and are mutable, they are like the ones that can be found in other programming languages. They are declared using the keyword *var* and assinged using =. Variables can have as many dimensions as needed and they only contain arithmetic expressions as values.

Functions define generic pieces of code that perform some computations in order to return arithmetic expressions. The only elements that can be declared inside them are

variables. They are defined as generic pieces of code because their paramenters and return value are not bound to a type, more about this will be said in the chapter about the type analysis that is implemented in the Circom 1.0 compiler.

```
Var: var name Dim
Function: function(args) block
```

Thanks to the mutability of variables the programmer can perform complex operations inside a template without breaking the restriction of only assigning once to a signal. Using the benefits of variables, the template in figure 2.2 can be generalized to any number of inputs. As can be seen in the following code, line 10 instantiantes the template to a concrete number of inputs, five in this case.

```
1   template Add(n) {
2       signal input vals[n];
3       signal output ret;
4       var s = 0;
5       for(var i = 0; i < n; i++) {
6           s += vals[i];
7       }
8       ret <-- s;
9   }
10  component main = Add(5);
```

Figure 2.3: Adder defined for any number of inputs

Some issues must be clarified about scoping in Circom before concluding this section. In Circom scoping works for variables just like it works in C or Rust. However signals and components must be declared at the initial scope of the template so the Circom program makes sense as a circuit.

## 2.2. Including the constraint set

As said earlier in this thesis, Circom programs not only define circuits but they also contain the executable specification needed for generating the constraint set of them. Circom allows the programmer to do this in a really simple way by including the operator '===='. Every time this operator appears the compiler will check that it is a constraint of the form $A * B - C = 0$ were A, B and C are linear combinations of signals. If the code in figure 2.3 where to generate its constraint set when given to the Circom 1.0 compiler, only one line needs to be added.

```
1   template Add(n) {
2       signal input vals[n];
3       signal output ret;
4       var s = 0;
5       for(var i = 0; i < n; i++) {
6           s += vals[i];
7       }
8       ret <-- s;
9       ret === s;
10  }
11  component main = Add(5);
```

Figure 2.4: Adder defined for any number of inputs with its constraint set

Only line 9 was added. In fact lines 8 and 9 represent a very common pattern in Circom so they can be reduced to just one line using the operators <== and ==>. Making the circuit in figure 2.3 generate its constraint set can be achieved by changing two characters.

```
1   template Add(n) {
2       signal input vals[n];
3       signal output ret;
4       var s = 0;
5       for(var i = 0; i < n; i++) {
6           s += vals[i];
7       }
8       ret <== s;
9   }
10  component main = Add(5);
```

Figure 2.5: Code in 2.4 simplified

As said at the start of this chapter, the meaning of signal tags was left for this section. Tags restrict what kind of values can be passed to a signal when using the operators <== and ==>. If no tag is specified or is FieldElement, then the signal accepts any kind values. When the Binary tag is used in a signal definition only other Binary tagged signals can be assigned to it through <== and ==>.

Variables and signals without tags are always considered FieldElements because Circom is parametric to a prime number. This implies that the arithmetic operations allowed in Circom work modulo this prime so the values stay in the field. The prime is set by default to: 21888242871839275222246405745257275088548364400416034343698204186575808495617.

Once the Circom 1.0 compiler processes the circuit in figure 2.5 the constraint generated will be: $ret - vals[0] - vals[1] - vals[2] - vals[3] - vals[4] = 0$. The same thing would happen if the circuit in figure 2.4 is processesed.

Constraints can appear anywhere inside a template and can depend on the values that some variables have at certain point of execution. This fact makes everything in a Circom program an executable specification. Before the constraint generation phase, the Circom 1.0 compiler makes sure that it is going to be capable of knowing the lengths of all the variables, signals and components in the program and that every constraint in the program will be reachable.

Chapter 3

# The Circom 1.0 compiler structure

This chapter is a bird's eye view of the Circom 1.0 compiler. Since this thesis will just explain in detail the parts that make the Circom 1.0 compiler special this chapter is a good way of summarizing all the implementation work done in this compiler.

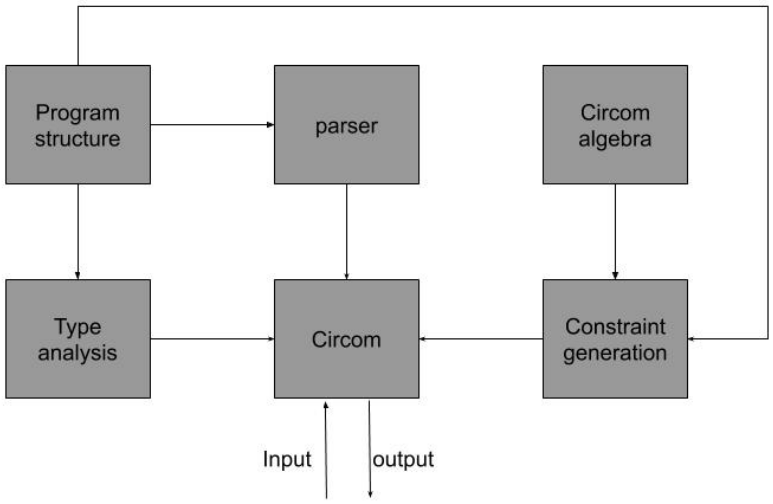



Figure 3.1: Bird's eye view of the Circom 1.0 compiler

In figure 3.1 the different modules that are working together in the Circom 1.0 compiler can be seen. The arrows connecting modules mean that one module (nock of the arrow) is beign used by another module (tip of the arrow). This chapter will provide a summary of these modules and how they interact with each other.

## 3.1. Program Structure

By looking into figure 3.1 it is not hard to notice that this module is used by every other one with the exception of the *Circom* module. Program structure defines the AST (abstract syntax tree) for the Circom programming language, which is widely used in the project. It also defines a data structure that gives quick access to useful program information, handles error reporting and defines useful constants like the prime number that generates the field. To conclude, this module contains a data structure that the rest of the modules can use to keep track of some program information while traversing the AST.

The AST parsed by the parser is really simple and extracting information from it requires traversing it. This module defines a data structure called *Program archive* that performs all the necessary tours through the AST and storages the useful information. Basically the *Program archive* makes working with the AST easier and more efficient.

Another useful data structure that is defined inside this module is the *Circom environment*. This data structure makes easier to keep track of the symbols that are in scope and the data assigned to them. This is useful because most of the static analyses this compiler performs and the constraint generation require keeping track of some information about each symbol.

## 3.2. Parser

This module contains all the code needed for parsing Circom programs. Circom programs can use functions and templates defined in other files using the "*include*" keyword. These inclusions are treated in the simplest possible way, all the definitions found in included files and in the main file are treated as if all of them were defined in the same file.

This module will load all the files that are part of the given program. The programmer does not have to worry about including the same file twice since this module will take care of not including already included files.

Once all the files are loaded this module will send them to the parser of the grammar. The grammar defined for Circom is a LR(1) grammar [18] built using a parser generator called lalrpop [11]. Once the grammar is written in the format required by lalrpop, this tool, despite its name, will prove that it is LR(1) and then it will produce the parser (Rust code [7]). Nothing more will be said about the grammar since building it was a matter of following the requirements of this tool.

If this module runs successfully then the *Program archive* for this progam will be generated. This module will report errors if a file in the program does not match the grammar, more than one main component is defined or the same name is used for defining several templates and functions.

## 3.3. Type analysis

Probably this name was not the best choice for encapsulating this module functionality since it does not stick only to the analysis of types. Several analyses are implemented inside this module, each one of them proving different properties of the program while traversing it.

In the comming chapters two of the analyses that can be found in this module will be explained, the type analysis and the unknown-known analysis. They are the ones that

required most of the work and they prove Circom-specific properties. These analyses are a great way of showing the kind of work done to build the Circom 1.0 compiler and getting to know the particularities of this programming language.

The tag analysis is also implemented in this module, as said in the previous chapter, tags are a way of restricting which elements can be assigned to a signal using ==> and <==. This analysis checks that those restrictions are met in the program.

A few analyses are implemented in this module to check that template elements (signals and components) are not declared inside functions and viceversa (templates do not use return statements). To conclude, one analysis is used to check that all the symbols are declared and used correctly which means that signals and components are declared at the initial scope, all the functions and templates that are called are defined in the program, the same symbol is not declared twice in the same scope and that all the symbols are in scope when accessed.

## 3.4. Constraint generation

The name of this module does not leave any room for imagination, it is the one in charge of executing the program and generating all the constraints. Since this execution is performed in a special way there is a chapter in this thesis that will explain it in detail.

## 3.5. Circom algebra

The Circom 1.0 compiler needs to store and modify artihmetic expressions during the constraint generation phase. These expressions are either written inline or stored in variables and computed as the program runs. As will be explained in the constraint generation chapter, there is no value stored inside signals during the constraint generation phase. Signals are terms in artihmetic expressions therefore their value does not matter.

This module handles all the operations that can be applied to arithmetic expressions. It distinguishes between linear expressions, quadratic expressions and non-quadratic expressions. Linear expressions are of the form $a_1X_1 + .... + a_nX_n$ where $a_{1..n}$ are numbers and $X_{1..n}$ signals. Quadratic expressions are of the form $A * B - C$ where $A$, $B$ and $C$ are linear expressions. Finally, non-quadratic expressions are those that can not be expressed as a linear expression nor as a quadratic expression.

This module also handles the fact that all the numbers in Circom belong to a field. In modular arithmetic some operations are not performed as usual so this module implements them. Usually programming languages have a library for performing such operations but the integers Circom handles are really big (remember the prime that generates the field). A library for performing modular arithmetic over big integers had to be written and can be found in this module.

## 3.6. Circom

Finally, the Circom module is the one that handles the interaction with the user and the flow of the Circom 1.0 compiler. When running the Circom 1.0 compiler users will provide the path to the Circom program and the options of their selection. For example the user can choose to specify the path where the compiler should place the output or the path where libraries are found. All the user demands are processed by this module and the compiler is executed according to them.

Chapter 4

# The type analysis

This chapter focuses on how the Circom 1.0 compiler proves that the types in a Circom program are correct. At the start of this chapter, this will look like an easy task, a matter of implementing an algorithm for type checking. Then the fact that templates and functions define generic pieces of code will be visited and things will get harder. This chapter will explain how the compiler deals with this scenario during the type analysis.

## 4.1. Signals, variables and components

Signals, like variables, belong to the arithmetic expression type which means that signals can be assigned to variables (and viceversa), the type analysis does not check signal tags, that work is done by another analysis. The next chapter will explain in detail the effect that signals have in arithemtic expressions. Signals and variables can be declared with as many dimensions as needed, but arrays must be homogeneus. Only arrays of variables can be accessed partially.

Typing components is trickier. Just like variables and signals they can be declared as arrays and, like signals, these arrays cannot be accessed partially. The type of a single component depends on the template to which it is initialized and it must be the same in every path of the code. The following code will pass the type checking.

```
1 var n = 12;
2 component a;
3 ... some code that modifies n ...
4 if(n > 0) { a = B(1,3);}
5 else { a = B(4,3); }
```

As can be seen in this example, the restriction over the type of the template does not include the parameters. Arrays of components must be homogeneus, all the components must be initialized to the same template but the parameters can be different. The type analysis will check that the signals accessed through a component are inputs and outputs of the template to which it was initialized.

## 4.2. Generic pieces of code

At this point the analysis looks quite easy, it needs to check that components are not assigned to signals or variables and viceversa, signal accesses in components match the type

of the component and array accesses in symbols match their dimensions. This analysis gets more complicated because templates and functions are generic pieces of code.

There is a restriction over how generic templates and functions can be. Components cannot be passed as arguments so functions and templates are generic to the number of dimensions their parameters have.

Two concepts should be defined before getting into the details of this analysis.

**Definition.** *Given a template $T$ with parameters $p_1...p_n$ a template type instance for $T$ is a vector of the form $[(p_1, a_1), ..., (p_n, a_n)]$, where $a_{1...n}$ are the number of dimensions assigned to the parameters $p_{1...n}$.*

**Definition.** *Given a function $F$ with parameters $p_1...p_n$ a function type instance for $F$ is a tuple of the form $(t_1, t_2)$. In this tuple $t_1$ is the vector $[(p_1, a_1), ..., (p_n, a_n)]$ being $a_{1...n}$ the number of dimensions assigned to the parameters $p_{1...n}$. The second element of the tuple, $t_2$, is the number of dimensions the return value will have when the arguments match $t_1$.*

This analysis finds for each template its template type instances and for each function its function type instances. When the type instances are found the type analysis just needs to check that the generic code is valid for each one of them. There are two scenarios where the type analysis will report an error, if a type instance cannot be created or if the type checking leads to an error.

In Circom programs there is just one statement that is not part of generic code, the declaration and initialization of the *main* component. This statement is the entry point for the type analysis, through it the first template type instance can be created. When given a type instance for some generic code, the type analysis guarantees that the types of all the symbols declared inside generic code are explicitly written in their declaration. This means that given some generic code and some type instance for it, the type analysis job is to check that the code matches the type instance and that new type instances created inside it match their respective generic code.

This analysis has the peculiarity of not traversing the code that is not used. This makes sense since generic code will only affect the execution of the program when instantiated. At the end, the cost of this analysis is the cost of traversing the code of functions and templates multiplied by the number of different type instances each of them have.

## 4.3. Creating type instances

Once a type instance for some generic code is created, knowing whether or not the code matches the type instance is relatively easy. It is a matter of implementing a type checking algorithm. Every time some generic code is called in a Circom program the type analysis must be able to infer the type instance that is being created. The following code will be an easy scenario for the type analysis.

```
1 function fun(p1,p2) {...}
2 template temp(p1,p2) {...}
3 var n[12] = fun(12,3);
4 component c = temp(n,4);
```

In this code line 3 creates the function type instance $([(p1, 0), (p2, 0)], 1)$ for *fun*. The dimensions of the return type are revealed by the declaration of *n*. Then, in line 4, a template type instance of the form $[(p1, 1), (p2, 0)]$ is created for *temp* and the type checking will be ran.

This type analysis also works if things get a little bit harder, like in the following example.

```
1 function fun(p1,p2) {...}
2 template temp(p1,p2) {...}
3 var n[12] = fun(12,3);
4 component c = temp(fun(7,1),4);
```

In this case line 3 stays the same so the analysis performs in the same way as before. Then in line 4 the analysis finds one difficulty, to infer the type instance that is being created first it needs to infer the type instance for the function call in the argument, $fun(7,1)$. This is not the worst case for the type analysis since it stores for each function and for each template all the different type instances that have been inferred already. The analysis can partially infer the type instance for the function call in line 4, since it knows the dimensions of the arguments. In this case the analysis will search through all the different type instances inferred for *fun* looking for one whose first element matches $[(p1,0),(p2,0)]$, since line three already checked a type instance that starts like that, the analysis infers that the resulting type has dimension 1.

Things can be more complex if the code looks like the one in the following example.

```
1 function fun(p1,p2) {...}
2 template temp(p1,p2) {...}
3 component c = temp(fun(7,1),4);
```

In this example the partial type instance for *fun* in line 4 will not be matched by any of *fun*'s type instances, the type analysis needs to look for other ways of infering the type instance. To clarify, the goal is to infer the type that is returned by a function knowing only the types of its parameters. To achieve this goal the type analysis will traverse the call graph orginated by the call whose type instance is trying to infer, this is done without visiting the same node twice. For each node in the call graph that is visited, the AST of the function will be traversed looking for return statements. When a return statement is found, the type analysis can find two scenarios. The best scenario is the one where the expression being returned is formed by numbers, variables and parameters. In this case, the returned type can be computed and the analysis finishes. A worst scenario is the one where the expression being returned contains a call to a function. In this case, the analysis looks in that function's AST trying to find a return type and if it was not found it continues the search in the AST that started the process.

To conclude two things must be clarified. First, when nested function calls are found this analysis will try to infer the type instances from the inside out. This is the only possible way since the returned types of the inner ones are parameter types for the outers. And second, and this is an important characteristic of this analysis, there are situations when the returned type of a function cannot be inferred. The following code is one of those situations.

```
1 function b() {
2     return function a();
3 }
4 function a() {
5     return b();
6 }
7 template temp(p1,p2) {...}
8 component c = temp(a(),4);
```

In this case the type analysis will try to infer the return type of function *a*, then the AST of all the nodes in the call graph will be traversed looking for a return statement that does not depend on a function call. This return statement will not be found. It is important to notice that in this situation the analysis does not get into an infinite computation, nodes in the call graph are visited just once. As it may be noticed, the analysis will find itself in this situation only when there is a function call in the program that starts an infinite computation, thus the type analysis will end and the compiler will report an error saying so.

Chapter 5

# The unknown-known analysis

In this chapter the unknown-known analysis, a static analysis [19], will be explained. This is one of the most interesting analysis that we have introduced in the Circom 1.0 compiler, it handles a unique characteristic of the Circom programming language. Designing an arithmetic circuit as a Circom program implies that it can be transformed into executable code. This characteristic, although useful, comes with one risk. Circom programmers can make the mistake of creating a circuit whose design depends on its execution. The following example shows the problem.

```
1 template A() {...}
2 template wrong(p1,p2) {
3     signal input in;
4     signal output out;
5     component c;
6     if(in > 3) { c = A();}
7     else { ... }
8     ...
9 }
```

This example is defining a circuit that will change during its own execution, the initialization of component *c* depends on the value given to signal *in*. Obviously this is wrong, the value of a signal is not known until the circuit is ran. This is not the only way in which the programmer can make the mistake of mixing design and execution, in fact this is a particular case of the problem.

The meaning of "designing a circuit" should be clarified. Obviously the declaration of signals and components is part of the circuits design, but this is covered because signals and components must be declared at the initial scope. There is one more element in a Circom program that serves the purpose of designing the circuit: the constraints. Using constraints, the programmer can express how the circuit behaves, a Circom program is designed correctly if its constraints can be reached without knowing the value of any signal. Some may notice that the example shown before is a particular case of constraints being unreachable, component *c* may include some constraints in the circuit and to reach them the value of signal *in* must be known. To summarize, the task of this analysis is to ensure that during the constraint generation phase all the constraints in the program can be reached.

During its execution, this analysis tags variables and signals as *unknown* and *known*, hence its name. Every signal is tagged as *unknown* and variables are tagged like the last expression that was assigned to them. To explain how this analysis works in the cleanest

way possible a simple version of Circom's AST will be used. First of all, the expressions this analysis deals with are the following ones.

**Number**

This symbol represents the expression formed by a single number.

**Signal**

The name of a signal does not matter to this analysis, it only needs to know that a signal appears in an expression. This symbol represents the expression formed by a signal.

**Variable**(name,acc)

In this case the expression is a variable with name *name*, its dimensions are accessed using the vector of expressions *acc*. For example, the expression $a[2][3]$, where $a$ is a variable, is represented as **Variable**(a,[Number,Number]).

**Array**(values)

Arrays can be declared inline, this symbol represents the expression formed by an inline array. It contains the vector of expressions that form the array.

**FunCall**(params)

This symbol represents a function call where *params* is a vector formed by the expressions passed as arguments.

**Op**($e_1$,$e_2$)

To conclude, this symbol represents that some operation is applied to expressions $e_1$ and $e_2$.

At this point a function that computes if an expression is unknown or known can be defined. This computation is performed by **Tag**(e,$\Gamma$), where $e$ is some expression and $\Gamma$ is the environment applied to the expression. $\Gamma$ is an assignment of the tags **U** and **K** (unknown and known respectively) to variable names. **Tag** returns **U** if the expression is unknown and **K** if it is known.

$$\textbf{Tag}(\textbf{Number},\Gamma) = \textbf{K}$$

Obviously, a expression formed by a number is known.

$$\textbf{Tag}(\textbf{Singal},\Gamma) = \textbf{U}$$

An expression formed by a signal is unknown.

$$\textbf{Tag}(\textbf{Variable}(\text{name,acc}),\Gamma) = \textbf{U}, \text{ if } \exists \text{ e. e} \in \text{acc} \rightarrow \textbf{Tag}(\text{e},\Gamma) = \textbf{U}$$
$$\textbf{Tag}(\textbf{Variable}(\text{name,acc}),\Gamma) = \Gamma[\text{name}], \text{ otherwise}$$

An expression formed by a variable is unknown if one of its indexes evaluates to unknown. Otherwise the expression's tag is the tag assigned to the variable in the environment.

$$\textbf{Tag}(\textbf{Array}(\text{values}),\Gamma) = \textbf{U}, \text{ if } \exists \text{ e. e} \in \text{values} \rightarrow \textbf{Tag}(\text{e},\Gamma) = \textbf{U}$$
$$\textbf{Tag}(\textbf{Array}(\text{values}),\Gamma) = \textbf{K}, \text{ otherwise}$$

An expression formed by an array is unknown if one of its expressions is unknown, otherwise it is known.

$$\textbf{Tag}(\textbf{FunCall}(\text{params}),\Gamma) = \textbf{U}, \text{ if } \exists \text{ e. e} \in \text{params} \rightarrow \textbf{Tag}(\text{e},\Gamma) = \textbf{U}$$
$$\textbf{Tag}(\textbf{FunCall}(\text{params}),\Gamma) = \textbf{K}, \text{ otherwise}$$

A function call evaluates to unknown if one of the arguments evaluates to unknown. Otherwise, since signals can not be declared inside functions, it evaluates to known.

$$\textbf{Tag}(\textbf{Op}(e_1,e_2),\Gamma) = \textbf{U}, \text{ if } \textbf{Tag}(e_1,\Gamma) = \textbf{U} \vee \textbf{Tag}(e_2,\Gamma) = \textbf{U}$$
$$\textbf{Tag}(\textbf{Op}(e_1,e_2),\Gamma) = \textbf{K}, \text{ otherwise}$$

Applying an operator to two expressions evaluates to unknown if one of them evaluates to unknown, otherwise it evaluates to known.

The statements used to explain this analysis are the following ones. They are a simplified version of the ones that can be found in the Circom 1.0 compiler AST so the behaviour of the analysis can be explained better.

**VariableDeclaration**(name,dim)

This statement represents a variable declared with name *name*, its dimensions are represented by the vector of expressions *dim*. For example the statement *var n[12][3]* is represented as **VariableDeclaration**(n,[Number,Number]). Signal and component declarations are omitted, they must be declared at the initial scope and the symbol analysis checks that no undeclared symbols are used.

**VariableAssign**(name,acc,e)

When this statement appears, it means that variable *name* is being accessed using as indexes the expressions in *acc*. Then the expression $e$ is assigned to the variable.

**Constraint**($e_1$,$e_2$)

This statement representes $e_1 === e_2$. Signal assignments using operators $<--$ and $-->$ are omitted since they do not have any effect in the analysis, signals are always unknown. Signal assignments using $<==$ and $==>$ are just syntactic sugar for a simple signal assignment followed by a constraint, for this analysis the assignment part is ommited, leaving just the constraint.

**ComponentInitialization**(name,acc,template,arguments)

This statement means that component *name* is being accessed using as indexes the expressions in *acc*. Then it is being initialized using a call to *template* where *arguments* is a vector of the expressions passed as arguments.

$$S_1;S_2$$

Sequence of statements, $S_1$ is followed by $S_2$.

**if** cond **then** $S_1$ **else** $S_2$
**while** cond **do** S
**Block**(S)

Representation of if-then-else, block and while loop statements. They are written together since by looking at the statements the meaning is quite clear.

The following functions over statements will be used in the analysis.

**declarations**(S) ≡ returns the set of symbols declared in statement S.
**assignments**(S) ≡ returns the set of symbols that receive a value in statement S.
**design**(S) ≡ returns if component is initialized or constraints are created in S.
**relevant**(S) ≡ **assignments**(S)/**declarations**(S).

Now the analysis is ready to be defined. The function **analyze** represents the behaviour of the analysis, it receives one of the previously defined statements and an environment. In the following lines we show how this function behaves for each type of statement. Also, this function returns either the environment resulting after the statement execution or an error.

```
1  fun analyze(VariableDeclaration(name,dim),Γ)
2      let Γ := Γ;
3      if ∃d.d ∈ dim → Tag(d,Γ) = U
4          ERROR
5      else
6          Γ[name] := U;
7          return Γ;
8
9  endfun
```

When a variable declaration is found, the variable declared is added to the environment tagged as unknown. An error will be reported if one of the dimensions is unknown, array lengths must be known during the constraint generation phase.

```
1  fun analyze(VariableAssign(name,acc,e),Γ)
2      let Γ := Γ;
3      if ∃d.d ∈ acc → Tag(d,Γ) = U
4          Γ[name] := U;
5      else
6          Γ[name] := Tag(e,Γ);
7      return Γ;
8  endfun
```

The assignment of an expression to a variable will make the variable unknown if one of its indexes is an unknown expression. Otherwise the variable will be assigned the tag of the expression.

```
1  fun analyze(ComponentInitialization(name,acc,template,arguments),Γ)
2      if ∃a.a ∈ arguments → Tag(a,Γ) = U
3          ERROR
4      else
5          return Γ;
6  endfun
```

A component initialization statement will report an error if there is one argument in the template call tagged as unknown.

```
1  fun analyze(Constraint(e1,e2),Γ)
2      return Γ;
3  endfun
```

In the actual implementation of this analysis, the compiler uses the environment to detect a special type of non-quadratic constraints. Constraints where some variable is accessed by an unknown index are non-quadratic.

```
1  fun analyze(S1;S2,Γ)
2      return analyze(S2,analyze(S1,Γ));
3  endfun
```

The environment returned when **analyze** is called over $S_1$ and $\Gamma$ is the one applied to $S_2$.

```
1  fun analyze(Block(S),Γ)
2      let Γ := Γ;
3      let Γ' := analyze(S,Γ);
4      forall name ∈ relevant(S)
5          Γ[name] := Γ'[name];
6      endforall
7      return Γ;
8  endfun
```

When a block statement is treated, first the environment resulting from analyzing the inner statements is computed. This environment is used to change in $\Gamma$ the tag of the variables that belong to the outer scope, which then will be returned.

```
1  fun analyze(if cond then S1 else S2,Γ)
2      if Tag(cond,Γ) = U ∧ design(S1)
3          ERROR
4      else if Tag(cond,Γ) = U ∧ design(S2)
5          ERROR
6      else
7          let Γ' := analyze(S1,Γ);
8          let Γ'' := analyze(S2,Γ);
9          forall name ∈ Γ'
10             if  Γ''[name] = U
11                 Γ'[name] := U;
12         endforall
13         return Γ'
14 endfun
```

In if-then-else statements if *cond* is tagged as unknown, the analysis must check that in any of the branches constraints are created or components are initialized. If this condition does not hold an error must be reported. To compute the resulting environment the analysis tags as unknown the symbols that became unknown in some of the branches.

```
1  fun analyze(while cond do S,Γ)
2      let Γ' := analyze(S,Γ);
3      let tag_1 := Tag(cond,Γ);
4      let tag_2 := Tag(cond,Γ');
5      if design(S) ∧ tag_1 = U
6          ERROR
7      else if design(S) ∧ tag_2 = U
8          ERROR
9      else
10         return Γ'
11 endfun
```

In while statements the analysis must check the tag of the condition for the initial environment and for the environment after one iteration of the loop. If the expression is tagged as unknown for one of the environments and constraints are generated or components are initialized in $S$, an error mut be reported.

This concludes the explanation of the unknown-known analysis. In reality, the Circom 1.0 compilers AST is more complex, but the important elements for this analysis are the ones captured in this chapter. As said before, this analysis is also used to detect some non-quadratic constraints before the constraint generation phase. This analysis is one of the most important elements in the Circom 1.0 compiler and captures a unique characteristic of Circom.

Chapter 6

# Constraint generation

During the constraint generation phase, the Circom 1.0 compiler computes the set of constraints that defines the behaviour of the circuit.

This phase of the Circom 1.0 compiler is splitted in two stages, in the first stage the Circom 1.0 compiler computes and stores all the constraints and in the second stage a simplification is applied to the set of constraints. There are Circom programs whose simplified output contains around one hundred million constraints, this means that time and memory efficiency during this phase is crucial.

This chapter explains how constraint generation and simplification is faced in the Circom 1.0 compiler and the improvements that have been made in terms of efficiency.

## 6.1. Symbolic execution

The Circom programs that enter this phase of the compiler will be executed, however their behaviour is not the same as the behaviour of compiled code. During this phase variables become holders of arithmetic expressions and signals are symbols in those expressions, hence the name of symbolic execution. This also means that all the operators behave as operators over arithmetic expressions. During this execution the compiler will distinguish between four types of arithmetic expressions.

- Constants: they are formed only by a number of the field.
- Linear: they are of the form $a_1 * s_1 + ... + a_n * s_n$ where $a_{1..n}$ are elements of the field and $p_{1..n}$ are signals. Additionaly, a term formed only by a field element can appear.
- Quadratic: they are of the form $A*B+C$ where $A$, $B$ and $C$ are arithmetic expressions of linear type.
- Non-quadratic: they cannot be expressed as an arithmetic expression of any of the other types.

Operations over arithmetic expressions can only make them increase in complexity, this means that when an arithmetic expression is non-quadratic there are no operations that can change its type. The only simplications done by the compiler are the ones that preserve the type of the expression. For example the arithmetic expression $2 * s_1 + 3 * s_1$ will be simplified to $5 * s_1$, but the expression $s_1 * s_2 - s_1 * s_2$ will not be simplified and will belong to the non-quadratic type.

The fact that in this phase variables are holders of arithmetic expressions becomes really useful when constraints involve a big number of signals, to ilustrate this the following example is provided.

```
1 ... template header ...
2 signal input in[5];
3 var expression = 0;
4 for(var i = 0; i < 5; i++) {
5     expression += in[i];
6 }
7 ... rest of statements ...
```

After the previous code is executed, the variable *expression* will contain the arithmetic expression $in[0] + in[1] + in[2] + in[3] + in[4]$ which is a linear arithmetic expression. In contrast, when this code is compiled to a target language and executed the inputs will have a concrete value and so the variable *expression*.

How the constraints generating statements are treated in this phase requires some explanation. Since signals are treated as symbols assignments to signals using the opertors $< --$ and $-- >$ can be ignored. Assignments using $<==$ and $==>$ can be treated as $===$, since the constraint generation behaviour of the statement is the only one that matters. With these simplifications, constraint statements can be simplified to those of the form $A_1 === A_2$ where $A_1$ and $A_2$ are arithmetic expressions of some type. When this statements are found the compiler will group both expressions at the same side of the equation, if the resulting arithmetic expression is of non-quadratic type, an error will be reported. Expressions of other types will be saved as arithmetic expressions of the form $A * B - C$ where $A$, $B$ and $C$ are linear arithmetic expressions. Constraints must be expressed this way because they represent equations of the form $A * B = C$.

## 6.2. Storing constraints and template sharing

During symbolic execution; Circom programs are represented as a tree of components with their respective constraints. Each node in this tree contains the name of the component created, its signals and the constraints that were created. Arrows between nodes appear in response to modular circuit design. Circuits can be created by connecting smaller circuits so in a Circom program the initialization of a component will trigger the initialization of its inner components. Arrows in this tree represent that relation between components. The following image shows how this tree looks like.

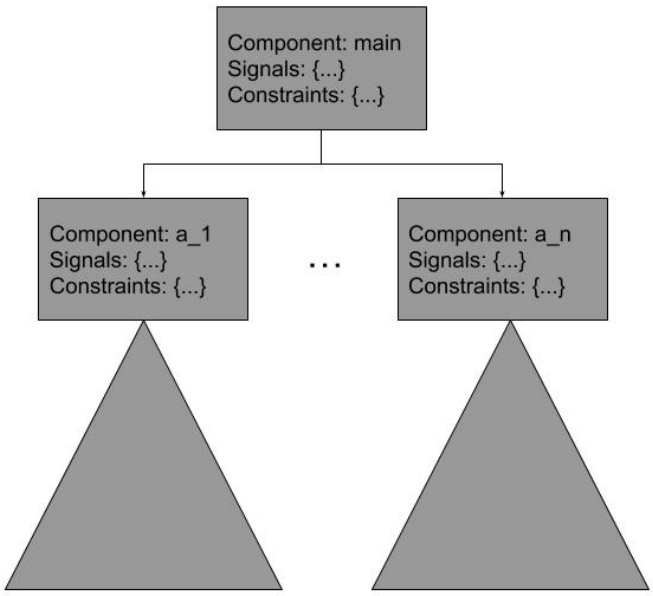



Figure 6.1: Circom circuit as a tree of components.

The tree in figure 6.1 can make us wonder if storing the names of the components and its signals is necessary. In fact, a set with all the constraints looks sufficient. To answer this question, the expected output of the Circom 1.0 compiler must first be understood. The output is a JSON file divided in two parts, the first part is an enumeration of all the signals in the circuit. In this part of the JSON the name of the signals can create ambiguity, there may be signals with the same name in different templates and different components can instantiate the same template. In order to erase this ambiguity, all the signal names in the JSON are written prefixed with a string formed by the names of the components that were created to get to the signal. In the second part of the JSON, the constraints of the circuit are written, but not as they were stored. The signal symbols in these constraints are changed to the numbers that were assigned to them in the first part. In order to create a final ouptut with these characteristics the tree structure is needed.

The tree in figure 6.1 is a good representation of the circuit execution in this phase. However, the Circom 1.0 compiler optimizes this structure to decrease the cost in memory and time, the concept behind these optimizations is called *template sharing*. This appeared after realising that in various Circom programs the same pattern appeared, in the following code this pattern can be seen.

```
1 ... template header ...
2 component c[size];
3 for (var i = 0; i < size; i++) {
4     c[i] = B(p0,p1);
5 }
6 ... some statements ..
```

In this code there is an array of components declared, after that all the components are initialized to the same template $B$ with the same arguments. Since the call to $B$ is receiving the same arguments in all the calls the subtrees generated by these calls are going to be identical. The Circom 1.0 compiler uses this fact to optimize the structure that its created in this phase. In this new structure, a node will represent a template call, it will contain the name of the template, the arguments used, the signals declared and the constraints that were created in the call. Arrows between nodes will now represent declarations of components inside templates and their corresponding initializations to a template with some arguments, this means that arrows now have a value attached to them, the name of the component.

As the reader may notice the structure created is not a tree anymore, now it is a DAG (directed acyclic graph). If in the scheme shown in figure 6.1 the components $a\_1$,...,$a\_n$ were to be initialized using the same template call, the DAG will look like in the following image.

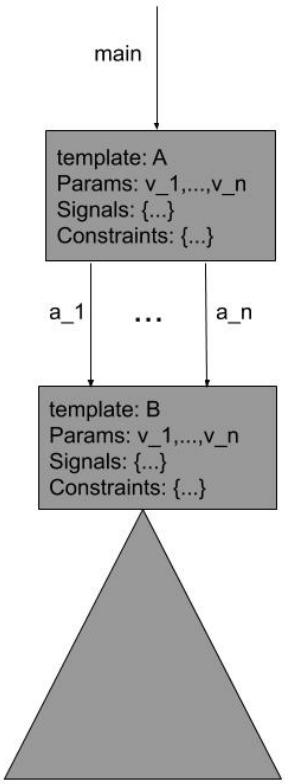



Figure 6.2: Constraint tree 6.1 optimized as a DAG.

Obviously the DAG is better in terms of memory efficiency, but also in terms of time efficiency. For each template call, the compiler will check if that template was already called using those arguments. If it is the case the compiler does not need to execute the template call, it only needs to create an arrow to the already existing node.

To test if Circom programs used in real life will obtain benefits from this optimization the rollup circuit [6] was used. The Circom program that tests this circuit has the following main component.

```
1 ... includes ...
2 component main = Rollup(5, 8);
```

When this Circom program is ran a DAG of 239 nodes is created and the final ouptut contains 814407 constraints. Then, if the parameters are changed to the ones in the following code the number of constraints increases to 35575741, but the number of nodes stays the same. This means that the execution of this Circom program takes a significant advantage of the template sharing, there are many component initializations were the template call was already executed.

```
1 ... includes ...
2 component main = Rollup(128, 24);
```

## 6.3. Constraint simplification

As the previous section explained, the data structure used to store all the information collected in this phase is more efficient thanks to the concept of template sharing. However, the cost of computing the JSON file remains the same as it would be without template sharing, the reason is that the constraints and signals of a template are written differently depending on the path of components that lead to the template. The only way to make this process more efficient is to reduce the number of constraints as much as possible.

In the Circom 1.0 compiler, an algorithm for reducing the number of constraints is implemented. It is called *local simplification* since it simplifies the constraint of each node in isolation, this algorithm visits each node in the DAG once. The local simplification works for each node in the following way.

1. Look in the set of constraints of the node for a linear constraint where some intermediate signal is part of a term. If there are none of them, stop.
2. Clear the intermediate signal of the constraint, save the resulting arithmetic expression and remove the constraint.
3. Subtitute in every constraint of the node the intermediate signal by the arithmetic expression obtained in step two.
4. Go back to step one.

In step one is important to notice that only linear constraints can be used. All the constraints in the DAG are of the form $A * B - C$, but if in the chosen constraint $A$ and $B$ are not empty the arithmetic expression obtained in step two is going to be non-quadratic. It is also important to notice that if the chosen constraint in step one is linear then the arithmetic expression obtained in step two is also linear. In this scenario the substitution performed in step three will maintain the type of all the constraints, since a signal is being changed by a linear expression.

When the second rollup test was executed applying this simplification, 31.280 constraints were removed of the final output, producing the JSON file took 5 minutes less. To conclude, it is worth acknowledging the fact that this simplification is done per node in the DAG (239 nodes in this example), which means that for most circuits this simplification has a very low impact in the Circom 1.0 compiler performance.

Chapter 7

# Experiments and evaluation

A series of experiments have been conducted to test the scalability and speed of the Circom 1.0 compiler. In this experiments a circuit is given to the Circom 1.0 compiler so it produces the output. When the compiler's execution ends the following measurements are collected:

- The compiler's execution time.
- The size of the JSON file.
- The number of constraints generated by the circuit.
- The number of signals generated by the circuit.

## 7.1. Rollup

In the context of blockchain, layer 2 refers to a secondary framework or protocol that is build on top of an existing blockchain. The main goal of these protocols is to solve the transaction speed and scaling difficulties that are faced by blockchain systems. The major examples of layer 2 solutions are the Bitcoin Lightning Network [13] and the Ethereum Plasma [12].

The zkRollup [6] is part of the iden3 project. This project is a layer 2 construction for the ethereum blockchain similar to Plasma. It uses the ethereum blockchain for data storage instead of computation, which is done off-chain. Although this thesis will not get into the behaviour of the zkRollup, there is a point in its execution where a zkSnarks is created. To achieve this, the rollup circuit is used and this is the one (with different instances) that will be used in these experiments.

The complexity of the templates used in the rollup circuit has helped us to test the perfomance of the symbolic execution. The use of template sharing in the Circom 1.0 compiler saves time by not executing template instances that have been executed before. In the rollup circuit there are templates formed by complex operations and, even if they are executed only once, is important to ensure that the symbolic execution is as fast as possible.

Another reason why the rollup circuit is used in these experiments is the number of constraints and signals it generates. To execute the rollup circuit a main component that instantiates a template called Rollup must be writen. This template receives two arguments whose value is directly proportional to the number of signals and constraints produced by

the circuit. As the reader will see in the next section, the rollup circuit generates millions of signals and constraints, even for small arguments.

To conclude, a good reason for using the rollup circuit in these experiments is that it is used in the real world. This implies that we know how the previous version of the Circom compiler performs with this circuit, so we can compare. Also, the rollup circuit is used by the community of iden3 and is part of an open-source project. This means that the improvements seen in this experiment will be experienced by the users of zkRollup.

## 7.2. Experiments

To obtain the previosly mentioned measurements a main component is provided to the rollup circuit. This component will instantiate the Rollup template giving as first argument a power of two and as second argument the value 24. The following table shows the results obtained for the rollup template when the Circom 1.0 compiler is ran without simplification.

| First argument | Second argument | Constraints | Signals | Time | Size |
|---|---|---|---|---|---|
| 16 | 24 | 4.666.511 | 4.666.602 | 0m 5,936s | 1.4 G |
| 32 | 24 | 9.118.239 | 9.118.714 | 1m 6,380s | 2.7 G |
| 64 | 24 | 18.021.695 | 18.022938 | 2m 12,475s | 5.5 G |
| 128 | 24 | 36.032.671 | 36.035.138 | 4m 34,301s | 11.1 G |
| 256 | 24 | 71.850.559 | 71.855.786 | 9m 4,260s | 22.3 G |
| 512 | 24 | 143.486.335 | 143.497.082 | 18m 48,634s | 44.8 G |
| 1024 | 24 | 286.757.887 | 286.779.674 | 37m 52,466s | 90.1 G |
| 2048 | 24 | 573.300.991 | 573.344.858 | 76m 5,850s | 180.9 G |

Table 7.1: Measurements obtained for the rollup circuit without simplification

As can be seen, as the arguments increase, the number of signals and constraints generated by the rollup circuit increase considerably. Now, to see the impact of the local simplification algorithm the same tests where runned, this time applying simplification. In the following table the reader can see the results obtained.

| First argument | Second argument | Constraints | Signals | Time | Size |
|---|---|---|---|---|---|
| 16 | 24 | 4.598.271 | 4.598.362 | 5m 53,815s | 1.4 G |
| 32 | 24 | 9.001.295 | 9.118.714 | 6m 28,919s | 2.7 G |
| 64 | 24 | 17.807.343 | 18.022.938 | 7m 51,128s | 5.5 G |
| 128 | 24 | 35.604.303 | 35.606.770 | 11m 2,019s | 11.1 G |
| 256 | 24 | 71.013.359 | 71.018.586 | 19m 32,888s | 22.2 G |
| 512 | 24 | 141.831.471 | 141.842.218 | 44m 28,208s | 44.6 G |
| 1024 | 24 | 283.467.695 | 283.489.482 | 125m 36,547s | 89.7 G |
| 2048 | 24 | 566.740.143 | 566.784.010 | 420m 9,966s | 180.2 G |

Table 7.2: Measurements obtained for the rollup circuit with simplification

## 7.3. Evaluation

When given the rollup circuit with parameters 2048 and 24 to the previous version of the Circom compiler it takes nearly a week to generate the results. As the reader can see the Circom 1.0 compiler only needs 76 minutes without simplification and 7 hours with simplification, this difference shows the impact that template sharing has in the perfomance of the Circom compiler.

Scalability is also a problem for the previous version of the compiler, for big circuits it needs to be runned in a system with many resources, otherwise the lack of memory will end the Compiler execution. The Circom 1.0 compiler scales much better, these experiments were run using a personal computer (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 6 cores, 16,2Gb de RAM).

As can be seen in tables 7.1 and 7.2, the local simplification algorithm reduces the number of signals and constraints but the cost in time is too high. At the time this thesis is written, the local simplification algorithm is being improved so less signals and constraints remain after its execution. In the previous version of Circom, when it is ran with simplification, although its time perfomance is much worse, it is able to remove more signals and constraints. We are studying new forms of simplification that can reduce the number of constraints drastically while keeping a similar efficiency to the simplification that we are currently applying.

Chapter 8

# Conclusions and Future Work

This thesis will conclude explaining why the Circom 1.0 compiler has been a great addition to the Circom project and how the knowledge obtained in the master on formal methods has helped to make it possible.

When the Circom 1.0 compiler project started, Circom already had a working compiler, Circom 0, and the version 0.5 was going to be released. Circom 0.5 is now working fine for those Circom programs that are known to be well written. However, when mistakes are made in Circom programs, the behaviour of this version is unpredictable. In this version some circuits with mistakes can pass all the checks done by the compiler, even if those mistakes affect the constraints of the circuit. This happens because the implementation was focus in the functionality Circom programmers want to have for their well written circuits, but not that much attention was payed into defining when a Circom program is well written.

The first thing that had to be done to avoid having the same problems in the Circom 1.0 compiler was to give a better definition of the language. Trying to achieve this, the restrictions over signal and components declarations and the unknown-known analysis were defined. These definitions have made Circom, as a programming language, stronger and the Circom 1.0 compiler represents the path to follow in future improvements. It is in this part of the project, that required thinking about Circom's semantics and structure, where the knowledge obtained in the master on formal methods has been essential. Knowing the importance of well-defined semantics in programming languages has been key to detect some incorrect circuits at an early state. In the unknown-known analysis the knowledge obtained in the master about static analysis was of great use. Thanks to this analysis, we can be sure that all the programs that enter the constraint generation phase will fulfill certain properties. This creates a scenario suitable for all the optimizations made in the constraint generation phase. As the reader can see, the knowledge provided by the master on formal methods has brought benefits that take effect throughout all the project.

The Circom 1.0 compiler has proven to be an improvement also when Circom programs are well written. The constraint generation phase of this version compared with the previous ones has been improved with the implementations of template sharing and simplification algorithms. Creating all the constraints takes less time and memory, which is really useful in this language since circuits can generate many contraints. Constraint simplification is also a great enhancement, writing the constraints to the required output is going to be costly in every version of the compiler unless there is a way of expressing the same with less data. In the Circom 1.0 compiler this is achieved by decreasing the

number of constraints that need to be written to obtain a smaller, but equivalent, set of constraints.

The following code is used to compare the performance of the Circom compiler version 0.5 and the Circom 1.0 compiler.

```
1  template Square() {
2      signal input in;
3      signal output out;
4
5      out <== in*in;
6  }
7  template Main(n) {
8      signal input in;
9      signal output out;
10
11     component squares[n];
12
13     var i;
14     for (i=0; i<n; i++) {
15         squares[i] = Square();
16         if (i==0) {
17             squares[i].in <== in;
18         } else {
19             squares[i].in <== squares[i-1].out;
20         }
21     }
22
23     squares[n-1].out ==> out;
24 }
```

Figure 8.1: Circuit for computing the square of each number in $1..n$

In this code there are two templates, Main and Square. Square computes the square of the value given as input and returns it through its output signal. Main receives the parameter $n$ when its instantiated and computes the squares of the numbers in $1..n$ using Square components. To execute the test a main component instantiating Main was written, the argument passed to this template was increased in each execution of the test.

The following table shows how much time each of the compilers took for each value of $n$. The first column shows the number passed as argument to the Main template. Then the time consumed by both versions is shown. To obtain these measurements the constraint simplification algorithm was not run in the Circom 1.0 compiler this way, since the number of constraints that have to be written is the same in both compilers, the benefits of template sharing will be clearer.

| n | Circom 0.5 | Circom 1 |
|---|---|---|
| 1000 | $0m0.518s$ | $0m0.079s$ |
| 10000 | $0m1.354s$ | $0m0.614s$ |
| 100000 | $0m9.755s$ | $0m5.851s$ |
| 1000000 | $2m54.727s^*$ | $1m6.826s$ |

Table 8.1: Time comparison of between Circom 0.5 and Circom 1 using figure 8.1

As can be seen in 8.1 , the Circom 1.0 compiler performed better than its previous version. The measurement obtained in the last test for Circom 0.5 is marked with an asterisk since the execution ended because of a memory error after that time.

## 8.1. Future work

Circom is part of the Iden3 project, which has a community of users and developers. This means that the Circom 1.0 compiler will be ready for distribution and production and, as the community asks for new functionalities, they will be implemented.

The following tasks are part of the future work that must done in the Circom 1.0 compiler.

- Implement the translation of Circom programs to a target language. At first the target language will be WebAssembly [15] but more languages will need to be targeted as time passes so this process must be built to be scalable.
- At this point the constraints are written in a JSON format. While the Circom 1.0 compiler was being built a binary format for writing the constraints of a Circom program was released. Constraints now can be written in r1cs files [5] which are smaller and writing them takes less time. In future iterations of the Circom 1.0 compiler this format for the constraints will be adopted.
- Integrating in the project a robust testing framework must be done at some point. As a project that will grow and change as time passes, the Circom 1.0 compiler must use some testing framework to keep the quality of the project stable.

As already said, different ways of improving the compiler and new functionalities will appear as the community starts to use it.

# Bibliography


[1] 0Kims. 0kims web page. `https://0kims.org`, 2020. [Online; accessed 26-June-2020].

[2] Eberhardt, J. and Tai, S. Zokrates - scalable privacy-preserving off-chain computations. 2018.

[3] iden3. Circom compiler 0.5. `https://github.com/iden3/circom`, 2020. [Online; accessed 26-June-2020].

[4] iden3. Iden3 web page. `https://iden3.io`, 2020. [Online; accessed 26-June-2020].

[5] iden3. r1csfile explanation. `https://github.com/iden3/r1csfile/blob/master/doc/r1cs_bin_format.md`, 2020. [Online; accessed 26-June-2020].

[6] iden3. Rollup circuit repository. `https://github.com/iden3/rollup`, 2020. [Online; accessed 26-June-2020].

[7] Klabnik, S. and Nichols, C. The Rust programming language. `https://doc.rust-lang.org/book/title-page.html`, 2020. [Online; accessed 26-June-2020].

[8] o1 labs. Snarky, an OCaml front-end for writing R1CS SNARKs. `https://github.com/o1-labs/snarky`, 2020. [Online; accessed 26-June-2020].

[9] matter labs. Zinc, github repository. `https://github.com/matter-labs/zinc`, 2020. [Online; accessed 26-June-2020].

[10] Labs, M. Documentation of Zinc, a DSL for writing R1CS SNARKs built in Rust. `https://zinc.matterlabs.dev/index.html`, 2020. [Online; accessed 26-June-2020].

[11] lalrpop. lalrpop, a parser generator for Rust. `https://github.com/lalrpop/lalrpop`, 2020. [Online; accessed 26-June-2020].

[12] Poon, J. Plasma : Scalable autonomous smart contracts. 2017.

[13] Poon, J. and Dryja, T. The bitcoin lightning network. 2016.

[14] Quisquater, J.-J., Quisquater, M., Quisquater, M., Quisquater, M., Guillou, L., Guillou, M. A., Guillou, G., Guillou, A., Guillou, G. and Guillou, S. How to explain zero-knowledge protocols to your children. In *Advances in Cryptology — CRYPTO' 89 Proceedings* (edited by G. Brassard), 628–631. Springer New York, New York, NY, 1990. ISBN 978-0-387-34805-6.

[15] Rossberg, A. WebAssembly specification. https://webassembly.github.io/spec/core/, 2020. [Online; accessed 26-June-2020].

[16] Torczon, L. and Cooper, K. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edn., 2007. ISBN 012088478X.

[17] Wikipedia. Wikipedia post about blockchain. https://en.wikipedia.org/wiki/Blockchain, 2020. [Online; accessed 26-June-2020].

[18] Wikipedia. Wikipedia post about LR parsers. https://en.wikipedia.org/wiki/LR_parser, 2020. [Online; accessed 26-June-2020].

[19] Wikipedia. Wikipedia post about static analysis. https://en.wikipedia.org/wiki/Static_program_analysis, 2020. [Online; accessed 26-June-2020].

[20] Wikipedia. Wikipedia post about zero-knowledge proofs. https://en.wikipedia.org/wiki/Zero-knowledge_proof, 2020. [Online; accessed 26-June-2020].

[21] Zokrates. Zokrates, github repository. https://github.com/Zokrates/ZoKrates, 2020. [Online; accessed 26-June-2020].