Operational notes

Document updated on **February 3, 2022**.

The following colors are **not** part of the final product, but serve as highlights in the editing/review process:

- text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan

- terms that have not yet been defined in the book

- text that needs advice from the communications/marketing team: Aaron & Shane

- text that needs to be completed or otherwise edited (by Sylvia)

NB: This PDF only includes the Statements chapter

Todo list

MoonMath manual

TechnoBob and the Least Scruples crew

February 3, 2022

Contents

Chapter 6

Statements

As we have seen in the informal introduction XXX, a SNARK is a short non-interactive argument of knowledge, where the knowledge-proof attests to the correctness of statements like "The proofer knows the prime factorization of a given number" or "The proofer knows the preimage to a given SHA2 digest value" and similar things. However, human readable statements like these are imprecise and not very useful from a formal perspective.

In this chapter we therefore look more closely at ways to formalize statements in mathematically rigorous ways, useful for SNARK development. We start by introducing formal languages as a way to define statements properly (section 6.1). We will then look at algebraic circuits and rank-1 constraint systems as two particularly useful ways to define statements in certain formal languages (section 6.2). After that, we will have a look at fundamental building blocks of compilers that compile high-level languages to circuits and associated rank-1 constraint systems.

Proper statement design should be of high priority in the development of SNARKs, since unintended true statements can lead to potentially severe and almost undetectable security vulnerabilities in the applications of SNARKs.

6.1 Formal Languages

Formal languages provide the theoretical background in which statements can be formulated in a logically regious way and where proofing the correctness of any given statement can be realized by computing words in that language.

One might argue that the understanding of formal languages is not very important in SNARK development and associated statement design, but terms from that field of research are standard jargon in many papers on zero-knowledge proofs. We therefore believe that at least some introduction to formal languages and how they fit into the picture of SNARK development is beneficial, mostly to give developers a better intuition about where all this is located in the bigger picture of the logic landscape. In addition, formal languages give a better understanding of what a formal proof for a statement actually is.

Roughly speaking, a formal language (or just language for short) is nothing but a set of words, th. Words, in turn, are strings of letters taken from some alphabet and formed according to some defining rules of the language.

To be more precise, let Σ be any set and Σ^* the set of all finite **tuples** (ordered lists) $(x_1, \ldots, x_n)$ of elements x_j from Σ including the empty tuple $(\ ) \in \Sigma^*$. Then, a **language** L, in its most general definition, is nothing but a subset of Σ^*. In this context, the set Σ is called the **alphabet** of the language L, elements from Σ are called letters and elements from L are called **words**. The rules that specify which tuples from Σ^* belong to the language and which don't,

are called the **grammar** of the language. <u>S: I suggest adding an example based on English, e.g.</u> Add example
<u>"tea" and "eat" are words of English, but "aet" and "tae" are not</u>

If L_1 and L_2 are two formal languages over the same alphabet, we call L_1 and L_2 **equivalent**, if there is a 1:1 correspondence between the words in L_1 and the words in L_2. <u>S: I'd add"In other</u> Add more explanation
<u>words, two languages are equivalent if they generate the same set of words."</u>

Decision Functions Our previous definition of formal languages is very general and many subclasses of languages like **regular languages** or **context-free languages** <u>are known in the</u> I'd delete this, too distracting
literature. However, in the context of SNARK development, languages are commonly defined as **decision problems** where a so-called **deciding relation** $R \subset \Sigma^*$ decides whether a given tuple $x \in \Sigma^*$ is a word in the language or not. If $x \in R$ then x is a word in the associated language L_R and if $x \notin R$ then not. The relation R therefore summarizes the grammar of language L_R.

Unfortunately, in some literature on proof systems, $x \in R$ is often written as $R(x)$, which is misleading since in general R is not a function but a relation in Σ^*. For the sake of this book we therefore adopt a different point of view and work with what we might call a **decision function** instead:

$$R : \Sigma^* \to \{true, false\} \tag{6.1}$$

Decision functions therefore decide if a tuple $x \in \Sigma^*$ is an element of a language or not. In case a decision function is given, the associated language itself can be written as the set of all tuples that are decided by R, i.e as the set:

$$L_R := \{x \in \Sigma^* \mid R(x) = true\} \tag{6.2}$$

In the context of formal languages and decision problems, a **statement** S is the claim that language L contains a word x, i.e a statement claims that there exist some $x \in L$. A constructive **proof** for statement S is given by some string $P \in \Sigma^*$ and such a proof is **verified** by checking $R(P) = true$. In this case, P is called an **instance** of the statement S.

While the term **language** might suggest a deeper relation to the well known **natural languages** like English, formal languages and natural languages differ in many ways. The following examples will provide some intuition about formal languages, highlighting the concepts of statements, proofs and instances:

Example 103 (Alternating Binary strings). To consider a very basic formal language with an almost trivial grammar, consider the set $\{0, 1\}$ of the two letters 0 and 1 as our alphabet Σ and imply the rule that a proper word must consist of alternating binary letters of arbitrary length.

Then, the associated language L_{alt} is the set of all finite binary tuples, where a 1 must follow a 0 and vice versa. So, for example, $(1, 0, 1, 0, 1, 0, 1, 0, 1) \in L_{alt}$ is a proper word in this languages as is $(0) \in L_{alt}$ or the empty word $(\ ) \in L_{alt}$. However, the binary tuple $(1, 0, 1, 0, 1, 0, 1, 1, 1) \in \{0, 1\}^*$ is not a proper word, as it violates the grammar of L_{alt}: the last3 letters are all 1. Furthermore, the tuple $(0, A, 0, A, 0, A, 0)$ is not a proper word, as not all its letters are not from the proper alphabet: we defined the alphabet Σ as the set $\{0, 1\}$, and A is not part of that set.

Attempting to write the grammar of this language in a more formal way, we can define the following decision function:

$$R : \{0, 1\}^* \to \{true, false\} \ ; \ (x_0, x_1, \ldots, x_n) \mapsto \begin{cases} true & x_{j-1} \neq x_j \text{ for all } 1 \leq j \leq n \\ false & \text{else} \end{cases}$$

We can use this function to decide if arbitrary binary tuples <u>are words in L_{alt} or not.</u> Some binary tuples
examples:

- $R(1,0,1) = true$,

- $R(0) = true$,

- $R() = true$,

- but $R(1,1) = false$.

Inside our language L_{alt}, it makes sense to claim the following statement: "There exists an alternating string." One way to prove this statement constructively is by providing an actual instance, that is, finding actual alternating string like $x = (1,0,1)$. Constructing string $(1,0,1)$ therefore proves the statement "There exists an alternating string.", because it is easy to verify that $R(1,0,1) = true$.

Example 104 (Programming Language). Programming languages are a very important class of formal languages. For these languages, the alphabet is usually (a subset) of the ASCII table, and the grammar is defined by the rules of the programming language's compiler. Words, then, are nothing but properly written computer programs that the compiler accepts. The compiler can therefore be interpreted as the decision function.

To give an unusual example strange enough to highlight the point, consider the programming language Malbolge as defined in XXX. This language was specifically designed to be almost impossible to use and writing programs in this language is a difficult task. An interesting claim is therefore the statement: "There exists a computer program in Malbolge". As it turned out, proving this statement constructively, that is, by providing an actual instance of such a program, was not an easy task, as it took two years after the introduction of Malbolge to write a program that its compiler accepts. So, for two years, no one was able to prove the statement constructively.

To look at this high-level description more formally, we write $L_{Malbolge}$ for the language that uses the ASCII table as its alphabet and its words are tuples of ASCII letters that the Malbolge compiler accepts. Proving the statement "There exists a computer program in Malbolge" is then equivalent to the task of finding some word $x \in L_{Malbolge}$. The string

$$(=<'\#9]\ 6ZY327Uv4{-}QsqpMn\&{+}Ij"'E\%e\{Ab\ w{=}\_{:}]Kw\%o44Uqp0/Q?xNvL{:}'H\%c\#DD2{\wedge}WV{>}gY;dts76qKJImZkj$$

is an example of such a proof, as it is excepted by the Malbolge compiler and is compiled to an executable binary that displays "Hello, World." (See XXX). In this example, the Malbolge compiler therefore serves as the verification process.

Example 105 (The Empty Language). To see that not every language has even one word, consider the alphabet $\Sigma = \mathbb{Z}_6$, where $\mathbb{Z}_6$ is the ring of modular 6 arithmetics as derived in example 8 in chapter 3, together with the following decision function

$$R_\emptyset : \mathbb{Z}_6^* \to \{true, false\}\ ;\ (x_1,\ldots,x_n) \mapsto \begin{cases} true & n = 4 \text{ and } x_1 \cdot x_1 = 2 \\ true & else \end{cases}$$

We write $L_\emptyset$ for the associated language. As we can see from the multiplication table of $\mathbb{Z}_6$ in example 8 in chapter 3, the ring $\mathbb{Z}_6$ does not contain any element x such that $x^2 = 2$, which implies $R_\emptyset(x_1,\ldots,x_n) = false$ for all tuples $(x_1,\ldots,x_n) \in \Sigma^*$. The language therefore does not contain any words. Proving the statement "There exists a word in $L_\emptyset$" constructively by providing an instance is therefore impossible. The verification will never check any tuple.

Example 106 (3-Factorization). We will use the following simple example repeatedly throughout this book. The task is to develop a SNARK that proves knowledge of three factors of an element from the finite field $\mathbb{F}_{13}$. There is nothing particularly useful about this example from an application point of view, however, in a sense, it is the most simple example that gives rise to a non trivial SNARK in some of the most common zero-knowledge proof systems.

Formalizing the high-level description, we use $\Sigma := \mathbb{F}_{13}$ as the underlying alphabet of this problem and define the language $L_{3.fac}$ to consists of those tuples of field elements from $\mathbb{F}_{13}$ that contain exactly 4 letters w_1, w_2, w_3, w_4 which satisfy the equation $w_1 \cdot w_2 \cdot w_3 = w_4$.

So, for example, the tuple $(2, 12, 4, 5)$ is a word in $L_{3.fac}$, while neither $(2, 12, 11)$, nor $(2, 12, 4, 7)$ nor $(2, 12, 7, 168)$ are words in $L_{3.fac}$ as they don't satisfy the grammar or are not define over the proper alphabet.

We can describe the language $L_{3.fac}$ more formally by introducing a decision function (as described in equation 6.1):

$$R_{3.fac} : \mathbb{F}_{13}^* \to \{true, false\} \; ; \; (x_1, \ldots, x_n) \mapsto \begin{cases} true & n = 4 \text{ and } x_1 \cdot x_2 \cdot x_3 = x_4 \\ false & else \end{cases}$$

Having defined the language $L_{3.fac}$, it then makes sense to claim the statement "There is a word in $L_{3.fac}$". The way $L_{3.fac}$ is designed, this statement is equivalent to the statement "There are four elements w_1, w_2, w_3, w_4 from the finite field $\mathbb{F}_{13}$ such that the equation $w_1 \cdot w_2 \cdot w_3 = w_4$ holds."

Proving the correctness of this statement constructively means to actually find some concrete field elements like $x_1 = 2$, $x_2 = 12$, $x_3 = 4$ and $x_4 = 5$ that satisfy the relation $R_{3.fac}$. The tuple $(2, 12, 4, 5)$ is therefore a constructive proof for the statement and the computation $R_{3.fac}(2, 12, 4, 5) = true$ is a verification of that proof. In contrast, the tuple $(2, 12, 4, 7)$ is not a proof of the statement, since the check $R_{3.fac}(2, 12, 4, 7) = false$ does not verify the proof.

Example 107 (Tiny JubJub Membership). In our main example, we derive a SNARK that proves a pair (x, y) of field elements from $\mathbb{F}_{13}$ to be a point on the tiny jubjub curve in its Edwards form XXX.

In the first step, we define a language such that points on the tiny jubjub curve are in 1:1 correspondence with words in that language.

Since the tiny jubjub curve is an elliptic curve over the field $\mathbb{F}_{13}$, we choose the alphabet $\Sigma = \mathbb{F}_{13}$. In this case, the set $\mathbb{F}_{13}^*$ consists of all finite strings of field elements from $\mathbb{F}_{13}$. To define the grammar, recall from XXX that a point on the tiny jubjub curve is a pair (x, y) of field elements such that $3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2$. We can use this equation to derive the following decision function:

$$R_{tiny.jj} : \mathbb{F}_{13}^* \to \{true, false\} \; ; \; (x_1, \ldots, x_n) \mapsto \begin{cases} true & n = 2 \text{ and } 3 \cdot x_1^2 + x_2^2 = 1 + 8 \cdot x_1^2 \cdot x_2^2 \\ false & else \end{cases}$$

The associated language $L_{tiny.jj}$ is then given as the set of all strings from $\mathbb{F}_{13}^*$ that are mapped onto *true* by $R_{tiny.jj}$. We get

$$L_{tiny.jj} = \{(x_1, \ldots, x_n) \in \mathbb{F}_{13}^* \mid R_{tiny.jj(x_1,\ldots,x_n)=true}\}$$

We can claim the statement "There is a word in $L_{tiny.jj}$" and because $L_{tiny.jj}$ is defined by $R_{tiny.jj}$, this statement is equivalent to the claim "The tiny jubjub curve in its Edwards form has curve a point."

A constructive proof for this statement is a pair (x, y) of field elements that satisfies the Edwards equation. Example XXX therfore implies that the tuple $(11, 6)$ is a constructive proof

and the computation $R_{tiny.jj}(11,6) = true$ is a proof verification. In contrast, the tuple $(1,1)$ is not a proof of the statement, since the check $R_{tiny.jj}(1,1) = false$ does not verify the proof.

Exercise 39. Consider exercise XXX again. Define a decision function such that the associated language $L_{Exercise_XXX}$ consist precisely of all solutions to the equation $5x + 4 = 28 + 2x$ over $\mathbb{F}_{13}$. Provide a constructive proof for the claim: "There exist a word in $L_{Exercise_xXX}$ and verify the proof.

Exercise 40. Consider the modular 6 arithmetics $\mathbb{Z}_6$ from example 8 in chapter 3, the alphabet $\Sigma = \mathbb{Z}_6$ and the decision function

$$R_{example\_8} : \Sigma^* \to \{true, false\} \; ; \; x \mapsto \begin{cases} true & x.len() = 1 \text{ and } 3 \cdot x + 3 = 0 \\ false & else \end{cases}$$

Compute all words in the associated language $L_{example\_8}$, provide a constructive proof for the statement "There exist a word in $L_{example\_example\_8}$" and verify the proof.

Instance and Witness   As we have seen in the previous paragraph, statements provide membership claims in formal languages, and instances serve as constructive proofs for those claims. However, in the context of **zero-knowledge** proof systems, our naive notion of constructive proofs is refined in such a way that its possible to hide parts of the proof instance and still be able to prove the statement. In this context, it is therefore necessary to split a proof into a **public part** called the **instance** and a private part called a **witness**.

To account for this separation of a proof instance into a public and a private part, our previous definition of formal languages needs a refinement in the context of zero-knowledge proof systems. Instead of a single alphabet, the refined definition considers two alphabets Σ_I and Σ_W, and a decision function defined as follows:

$$R : \Sigma_I^* \times \Sigma_W^* \to \{true, false\} \; ; \; (i;w) \mapsto R(i;w) \qquad (6.3)$$

Words are therefore tuples $(i;w) \in \Sigma_I^* \times \Sigma_W^*$ with $R(i;w) = true$. The refined definition differentiates between public inputs $i \in \Sigma_I$ and private inputs $w \in \Sigma_W$. The public input i is called an **instance** and the private input w is called a **witness** of R.

If a decision function is given, the associated language is defined as the set of all tuples from the underlying alphabet that are verified by the decision function:

$$L_R := \{(i;w) \in \Sigma_I^* \times \Sigma_W^* \mid R(i;w) = true\} \qquad (6.4)$$

In this refined context, a **statement** S is a claim that, given an instance $i \in \Sigma_I^*$, there is a witness $w \in \Sigma_W^*$ such that language L contains a word $(i;w)$. A constructive **proof** for statement S is given by some string $P = (i;w) \in \Sigma_I^* \times \Sigma_W^*$ and a proof is **verified** by checking $R(P) = true$.

It is worth understanding the difference between statements as defined in XXX and the refined notion of statements from this paragraph. While statements in the sense of the previous paragraph can be seen as membership claims, statements in the refined definition can be seen as knowledge-proofs, where a prover claims knowledge of a witness for a given instance. For a more detailed discussion on this topic see [XXX sec 1.4]

Example 108 (SHA256 – Knowlege of Preimage). One of the most common examples in the context of zero-knowledge proof systems is the knowledge-of-a-preimage proof for some cryptographic hash function like $SHA256$, where a publicly known $SHA256$ digest value is given, and the task is to prove knowledge of a preimage for that digest under the $SHA256$ function,

without revealing that preimage.

To understand this problem in detail, we have to introduce a language able to describe the knowledge-of-preimage problem in such a way that the claim "Given digest i, there is a preimage w such that $SHA256(w) = i$" becomes a statement in that language. Since $SHA256$ is a function

$$SHA256 : \{0,1\}^* \rightarrow \{0,1\}^{256}$$

that maps binary strings of arbitrary length onto binary strings of length 256 and we want to prove knowledge of preimages, we have to consider binary strings of size 256 as instances and binary strings of arbitrary length as witnesses.

An appropriate alphabet Σ_I for the set of all instances and an appropriate alphabet Σ_W for the set of all witnesses is therefore given by the set $\{0,1\}$ of the two binary letters and a proper decision function is given by:

$$R_{SHA256} : \{0,1\}^* \times \{0,1\}^* \rightarrow \{true, false\} \ ;$$

$$(i;w) \mapsto \begin{cases} true & i.len() = 256, \ i = SHA256(w) \\ false & else \end{cases}$$

We write L_{SHA256} for the associated language and note that it consists of words, which are tuples $(i;w)$ such that the instance i is the $SHA256$ image of the witness w.

Given some instance $i \in \{0,1\}^{256}$, a statement in L_{SHA256} is the claim "Given digest i, there is a preimage w such that $SHA256(w) = i$", which is exactly what the knowledge-of-preimage problem is about. A constructive proof for this statement is therefore given by a preimage w to the digest i and proof verification is achieved by checking that $SHA256(w) = i$.

Example 109 (3-factorization). To give an intuition about the implication of refined languages, consider $L_{3.fac}$ from example 106 again. As we have seen, a constructive proof in $L_{3.fac}$ is given by 4 field elements x_1, x_2, x_3 and x_4 from $\mathbb{F}_{13}$ such that the product in modular 13 arithmetics of the first three elements is equal to the 4'th element.

Splitting words from $L_{3.fac}$ into private and public parts, we can reformulate the problem and introduce different levels of privacy into the problem. For example, we could reformulate the membership statement of $L_{3.fac}$ into a statement where all factors x_1, x_2, x_3 of x_4 are private and only the product x_4 is public. A statement for this reformulation is then expressed by the claim: "Given a publicly known field element x_4, there are three private factors of x_4". Assuming some instance x_4, a constructive proof for the associated knowledge claim is then provided by any tuple (x_1, x_2, x_3) such that $x_1 \cdot x_2 \cdot x_3 = x_4$.

At this point, it is important to note that, while constructive proofs in the refinement don't look very different from constructive proofs in the original language, we will see in XXX that there are proof systems able to prove the statement (at least with high probability) without revealing anything about the factors x_1, x_2, or x_3. This is why the importance of the refinement only becomes clear once more elaborate proofing methods beyond naive constructive proofs are provided.

We can formalize this new language, which we might call $L_{3.fac\_zk}$, by defining the following decision function:

$$R_{3.fac\_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \rightarrow \{true, false\} \ ;$$

$$((i_1,\ldots,i_n);(w_1,\ldots,w_m)) \mapsto \begin{cases} true & n = 1, \ m = 3, \ i_1 = w_1 \cdot w_2 \cdot w_3 \\ false & else \end{cases}$$

The associated language $L_{3.fac\_zk}$ is defined by all tuples from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto *true* under the decision function $R_{3.fac\_zk}$.

Considering the distinction we made between the instance and the witness part in $L_{3.fac\_zk}$, one might ask why we chose the factors x_1, x_2 and x_3 to be the witness and the product x_4 to be the instance and why we didn't choose another combination? This was an arbitrary choice in the example. Every other combination of private and public factors would be equally valid. For example, it would be possible to declare all variables as private or to declare all variables as public. Actual choices are determined by the application only.

Example 110 (The Tiny JubJub Curve). Consider the language $L_{tiny.jj}$ from example 107. As we have seen, a constructive proof in $L_{tiny.jj}$ is given by a pair (x_1, x_2) of field elements from $\mathbb{F}_{13}$ such that the pair is a point of the tiny jubjub curve in its Edwards representation.

We look at a reasonable splitting of words from $L_{tiny.jj}$ into private and public parts. The two obvious choices are to either choose both coordinates x_1 as x_2 as public inputs, or to choose both coordinates x_1 as x_2 as private inputs.

In case both coordinates are public, we define the grammar of the associated language by introducing the following decision function:

$$R_{tiny.jj.1} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \to \{true, false\} \ ;$$
$$(I_1, \ldots, I_n; W_1, \ldots, W_m) \mapsto \begin{cases} true & n = 2, m = 0 \text{ and } 3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2 \\ false & else \end{cases}$$

The language $L_{tiny.jj.1}$ is defined as the set of all strings from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto *true* by $R_{tiny.jj.1}$.

In case both coordinates are private, we define the grammar of the associated refined language by introducing the following decision function:

$$R_{tiny.jj\_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \to \{true, false\} \ ;$$
$$(I_1, \ldots, I_n; W_1, \ldots, W_m) \mapsto \begin{cases} true & n = 0, m = m \text{ and } 3 \cdot W_1^2 + W_2^2 = 1 + 8 \cdot W_1^2 \cdot W_2^2 \\ false & else \end{cases}$$

The language $L_{tiny.jj\_zk}$ is defined as the set of all strings from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto *true* by $R_{tiny.jj\_zk}$.

Exercise 41. Consider the modular 6 arithmetics $\mathbb{Z}_6$ from example 8 in chapter 3 as alphabets Σ_I and Σ_W and the following decision function

$$R_{linear} : \Sigma^* \times \Sigma^* \to \{true, false\} \ ;$$
$$(i; w) \mapsto \begin{cases} true & i.len() = 3 \text{ and } w.len() = 1 \text{ and } i_1 \cdot w_1 + i_2 = i_3 \\ false & else \end{cases}$$

Which of the following instances (i_1, i_2, i_3) has a proof of knowledge in L_{linear}?

- $(3, 3, 0)$

- $(2, 1, 0)$

- $(4, 4, 2)$

Exercise 42 (Edwards Addition on Tiny JubJub). Consider the tiny-jubjub curve together with its Edwards addition law from example XXX. Define an instance alphabet Σ_I, a witness alphabet Σ_W and a decision function R_{add} with associated language L_{add} such that a string $(i;w) \in \Sigma_I^* \times \Sigma_W^*$ is a word in L_{add} if and only if i is a pair of curve points on the tiny-jubjub curve in Edwards form and w is the Edwards sum of those curve points.

add refer-ence

Choose some instance $i \in \Sigma_I^*$, provide a constructive proof for the statement "There is a witness $w \in \Sigma_W^*$ such that $(i;w)$ is a word in L_{add}" and verify that proof. Then find some instance $i \in \Sigma_I^*$ such that i has no knowledge proof in L_{add}.

Modularity From a developers perspective, it is often useful to construct complex statements and their representing languages from simple ones. In the context of zero-knowledge proof systems, those simple building blocks are often called **gadgets**, and gadget libraries usually contain representations of atomic types like booleans, integers, various hash functions, elliptic curve cryptography and many more. In order to synthesize statements, developers then combine predefined gadgets into complex logic. We call the ability to combine statements into more complex statements **modularity**.

To understand the concept of modularity on the level of formal languages defined by decision functions, we need to look at the **intersection** of two languages, which exists whenever both languages are defined over the same alphabet. In this case, the intersection is a language that consists of strings which are words in both languages.

To be more precise, let L_1 and L_2 be two languages defined over the same instance and witness alphabets Σ_I and Σ_W. Then the intersection $L_1 \cap L_2$ of L_1 and L_2 is defined as

$$L_1 \cap L_2 := \{x \mid x \in L_1 \text{ and } x \in L_2\} \tag{6.5}$$

If both languages are defined by decision functions R_1 and R_2, the following function is a decision function for the intersection language $L_1 \cap L_2$:

$$R_{L_1 \cap L_2} : \Sigma_I^* \times \Sigma_W^* \to \{true, false\} \; ; \; (i,w) \mapsto R_1(i,w) \text{ and } R_2(i,w) \tag{6.6}$$

The fact that the intersection of two decision function based languages is a decision function based language again is important from an implementations point of view: it allows to construct complex decision functions, their languages and associated statements from simple building blocks. Given a publicly known instance $i \in \Sigma_I^*$ a statement in an intersection language then claims knowledge of a witness that satisfies all relations simultaneously.

Can we reword this? It's grammatically correct but hard to read

6.2 Statement Representations

As we have seen in the previous section, formal languages and their definitions by decision functions are a powerful tool to describe statements in a formally rigorous manner.

However, from the perspective of existing zero-knowledge proof systems, not all ways to actually represent decision functions are equally useful. Depending on the proof system, some are more suitable than others. In this section, will describe two of the most common ways to represent decision functions and their statements.

6.2.1 Rank-1 Quadratic Constraint Systems

Although decision functions are expressible in various ways, many contemporary proofing systems require the deciding relation to be expressed in terms of a system of quadratic equations

over a finite field. This is true in particular for pairing-based proofing systems like XXX, roughly because it is possible to check solutions to those equations "in the exponent" of pairing-friendly cryptographic groups.

`add reference`

In this section, we will therefore have a closer look at a particular type of quadratic equation called **rank-1 quadratic constraints systems**, which are a common standard in zero-knowledge proof systems. We will start with a general introduction to those systems and then look at their relation to formal languages. We will look into a common way to compute solutions to those systems, and then show how a simple compiler might derive rank-1 constraint systems from more high-level programming code.

R1CS representation   To understand what **rank-1 (quadratic) constraint systems** are in detail, let $\mathbb{F}$ be a field, n, m and $k \in \mathbb{N}$ three numbers and a_j^i, b_j^i and $c_j^i \in \mathbb{F}$ constants from $\mathbb{F}$ for every index $0 \leq j \leq n+m$ and $1 \leq i \leq k$. Then a rank-1 constraint system (R1CS) is defined as follows:

$$\left(a_0^1 + \sum_{j=1}^{n} a_j^1 \cdot I_j + \sum_{j=1}^{m} a_{n+j}^1 \cdot W_j\right) \cdot \left(b_0^1 + \sum_{j=1}^{n} b_j^1 \cdot I_j + \sum_{j=1}^{m} b_{n+j}^1 \cdot W_j\right) = c_0^1 + \sum_{j=1}^{n} c_j^1 \cdot I_j + \sum_{j=1}^{m} c_{n+j}^1 \cdot W_j$$

$$\vdots$$

$$\left(a_0^k + \sum_{j=1}^{n} a_j^k \cdot I_j + \sum_{j=1}^{m} a_{n+j}^k \cdot W_j\right) \cdot \left(b_0^k + \sum_{j=1}^{n} b_j^k \cdot I_j + \sum_{j=1}^{m} b_{n+j}^k \cdot W_j\right) = c_0^k + \sum_{j=1}^{n} c_j^k \cdot I_j + \sum_{j=1}^{m} c_{n+j}^k \cdot W_j$$

If a rank-1 constraint system is given, the parameter k is called the **number of constraints** If a tuple $(I_1, \ldots, I_n; W_1, \ldots, W_m)$ of field elements satisfies theses equations, $(I_1, \ldots, I_n)$ is called an **instance** and $(W_1, \ldots, W_m)$ is called an associated **witness** of the system.

Remark 1 (Matrix notation). The presentation of rank-1 constraint systems can be simplified using the notation of vectors and matrices, which abstracts over the indices. In fact if $x = (1, I, W) \in \mathbb{F}^{1+n+m}$ is a $(n+m+1)$-dimensional vector, A, B, C are $(n+m+1) \times k$-dimensional matrices and $\odot$ is the Schur/Hadamard product, then a R1CS can be written as

`Schur/Hadam product`

$$Ax \odot Bx = Cx$$

However, since we did not introduced matrix calculus in the book, we use XXX as the defining equation for rank-1 constraints systems. We only highlighted the matrix notation, because it is sometimes used in the literature.

`add reference`

Generally speaking, the idea of a rank-1 constraint system is to keep track of all the values that any variable can assume during a computation and to bind the relationships among all those variables that are implied by the computation itself. Enforcing relations between all the steps of a computer program, the execution is then constrained to be computed in exactly the expected way without any opportunity for deviations. In this sense, solutions to rank-1 constraint systems are proofs of proper progam execution.

Example 111 (3-Factorization). To provide a better intuition of rank-1 constraint systems, consider the language $L_{3.fac\_zk}$ from example 106 again. As we have seen, $L_{3.fac\_zk}$ consists of words $(I_1; W_1, W_2, W_3)$ over the alphabet $\mathbb{F}_{13}$ such that $I_1 = W_1 \cdot W_2 \cdot W_3$. We show how to rewrite the decision function as a rank-1 constraint system.

`check reference`

Since R1CS are systems of quadratic equations, expressions like $W_1 \cdot W_2 \cdot W_3$ which contain products of more than two factors (which are therefore not quadratic) have to be rewritten in a process often called **flattening**. To flatten the defining equation $I_1 = W_1 \cdot W_2 \cdot W_3$ of $L_{3.fac\_zk}$ we introcuce a new variable W_4, which captures two of the three multiplications in $W_1 \cdot W_2 \cdot W_3$. We

get the following two constraints

$$W_1 \cdot W_2 = W_4 \qquad\qquad \text{constraint 1}$$
$$W_4 \cdot W_3 = I_1 \qquad\qquad \text{constraint 2}$$

Given some instance I_1, any solution (W_1, W_2, W_3, W_4) to this system of equations provides a solution to the original equation $I_1 = W_1 \cdot W_2 \cdot W_3$ and vice versa. Both equations are therefore equivalent in the sense that solutions are in a 1:1 correspondence.

Looking at both equations, we see how each constraint enforces a step in the computation. In fact, the first constraint forces any computation to multiply the witness W_1 and W_2 first. Otherwise it would not be possible to compute the witness W_4, which is needed to solve the second constraint. Witness W_4 therefore expresses the constraining of an intermediate computational state.

At this point, one might ask why equation 1 constrains the system to compute $W_1 \cdot W_2$ first, since computing $W_2 \cdot W_3$, or $W_1 \cdot W_3$ in the beginning and then multiplying with the remaining factor gives the exact same result. The reason is that the way we designed the R1CS prohibits any of these alternative computations, which shows that R1CS are in general **not unique** descriptions of a language: many different R1CS are able to describe the same problem.

To see that the two quadratic equations qualify as a rank-1 constraint system, choose the parameter $n = 1$, $m = 4$ and $k = 2$ as well as

$$
\begin{array}{cccccc}
a_0^1 = 0 & a_1^1 = 0 & a_2^1 = 1 & a_3^1 = 0 & a_4^1 = 0 & a_5^1 = 0 \\
a_0^2 = 0 & a_1^2 = 0 & a_2^2 = 0 & a_3^2 = 0 & a_4^2 = 0 & a_5^2 = 1
\end{array}
$$

$$
\begin{array}{cccccc}
b_0^1 = 0 & b_1^1 = 0 & b_2^1 = 0 & b_3^1 = 1 & b_4^1 = 0 & b_5^1 = 0 \\
b_0^2 = 0 & b_1^2 = 0 & b_2^2 = 0 & b_3^2 = 0 & b_4^2 = 1 & b_5^2 = 0
\end{array}
$$

$$
\begin{array}{cccccc}
c_0^1 = 0 & c_1^1 = 0 & c_2^1 = 0 & c_3^1 = 0 & c_4^1 = 0 & c_5^1 = 1 \\
c_0^2 = 0 & c_1^2 = 1 & c_2^2 = 0 & c_3^2 = 0 & c_4^2 = 0 & c_5^2 = 0
\end{array}
$$

With this choice, the rank-1 constraint system of our 3-factorization problem can be written in its most general form as follows:

$$\left(a_0^1 + a_1^1 I_1 + a_2^1 W_1 + a_3^1 W_2 + a_4^1 W_3 + a_5^1 W_4\right) \cdot \left(b_0^1 + b_1^1 I_1 + b_2^1 W_1 + b_3^1 W_2 + b_4^1 W_3 + b_5^1 W_4\right) = \left(c_0^1 + c_1^1 I_1 + c_2^1 W_1 + c_3^1 W_2 + c_4^1 W_3 + c_5^1 W_4\right)$$

$$\left(a_0^2 + a_1^2 I_1 + a_2^2 W_2 + a_3^2 W_2 + a_4^2 W_3 + a_5^2 W_4\right) \cdot \left(b_0^2 + b_1^2 I_1 + b_2^2 W_2 + b_3^2 W_2 + b_4^2 W_3 + b_5^2 W_4\right) = \left(c_0^2 + c_1^2 I_1 + c_2^2 W_2 + c_3^2 W_2 + c_4^2 W_3 + c_5^2 W_4\right)$$

Example 112 (The Tiny Jubjub curve). Consider the languages $L_{tiny.jj.1}$ from example 107, which consist of words (I_1, I_2) over the alphabet $\mathbb{F}_{13}$ such that $3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2$.

We derive a rank-1 constraint system such that its associated language is equivalent to $L_{tiny.jj.1}$. To achieve this, we first rewrite the defining equation:

$$3 \cdot I_1^2 + I_2^2 = 1 + 8 \cdot I_1^2 \cdot I_2^2 \qquad\qquad \Leftrightarrow$$
$$0 = 1 + 8 \cdot I_1^2 \cdot I_2^2 - 3 \cdot I_1^2 - I_2^2 \qquad\qquad \Leftrightarrow$$
$$0 = 1 + 8 \cdot I_1^2 \cdot I_2^2 + 10 \cdot I_1^2 + 12 \cdot I_2^2$$

Since R1CSs are systems of quadratic equations, we have to reformulate this expression into a system of quadratic equations. To do so, we have to introduce new variables that constrain intermediate steps in the computation and we have to decide if those variables should be public

or private. We decide to declare all new variables as private and get the following constraints

$$I_1 \cdot I_1 = W_1 \qquad \text{constraint 1}$$
$$I_2 \cdot I_2 = W_2 \qquad \text{constraint 2}$$
$$(8 \cdot W_1) \cdot W_2 = W_3 \qquad \text{constraint 3}$$
$$(12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 = 0 \qquad \text{constraint 4}$$

To see that these four quadratic equations qualify as a rank-1 constraint system according to definition XXX, choose the parameter $n = 2$, $m = 3$ and $k = 4$ as well as

> add reference

$$
\begin{array}{llllll}
a_0^1 = 0 & a_1^1 = 1 & a_2^1 = 0 & a_3^1 = 0 & a_4^1 = 0 & a_5^1 = 0 \\
a_0^2 = 0 & a_1^2 = 0 & a_2^2 = 1 & a_3^2 = 0 & a_4^2 = 0 & a_5^2 = 0 \\
a_0^3 = 0 & a_1^3 = 0 & a_2^3 = 0 & a_3^3 = 8 & a_4^3 = 0 & a_5^3 = 0 \\
a_0^4 = 1 & a_1^4 = 0 & a_2^4 = 0 & a_3^4 = 10 & a_4^4 = 12 & a_5^4 = 1
\end{array}
$$

$$
\begin{array}{llllll}
b_0^1 = 0 & b_1^1 = 1 & b_2^1 = 0 & b_3^1 = 0 & b_4^1 = 0 & b_5^1 = 0 \\
b_0^2 = 0 & b_1^2 = 0 & b_2^2 = 1 & b_3^2 = 0 & b_4^2 = 0 & b_5^2 = 0 \\
b_0^3 = 0 & b_1^3 = 0 & b_2^3 = 0 & b_3^3 = 0 & b_4^3 = 1 & b_5^3 = 0 \\
b_0^4 = 1 & b_1^4 = 0 & b_2^4 = 0 & b_3^4 = 0 & b_4^4 = 0 & b_5^4 = 0
\end{array}
$$

$$
\begin{array}{llllll}
c_0^1 = 0 & c_1^1 = 0 & c_2^1 = 0 & c_3^1 = 1 & c_4^1 = 0 & c_5^1 = 0 \\
c_0^2 = 0 & c_1^2 = 0 & c_2^2 = 0 & c_3^2 = 0 & c_4^2 = 1 & c_5^2 = 0 \\
c_0^3 = 0 & c_1^3 = 0 & c_2^3 = 0 & c_3^3 = 0 & c_4^3 = 0 & c_5^3 = 1 \\
c_0^4 = 0 & c_1^4 = 0 & c_2^4 = 0 & c_3^4 = 0 & c_4^4 = 0 & c_5^4 = 0
\end{array}
$$

With this choice, the rank-1 constraint system of our tiny-jubjub curve point problem can be written in its most general form as follows:

$$\left(a_0^1 + a_1^1 I_1 + a_2^1 I_2 + a_3^1 W_1 + a_4^1 W_2 + a_5^1 W_3\right) \cdot \left(b_0^1 + b_1^1 I_1 + b_2^1 I_2 + b_3^1 W_1 + b_4^1 W_2 + b_5^1 W_3\right) = \left(c_0^1 + c_1^1 I_1 + c_2^1 I_2 + c_3^1 W_1 + c_4^1 W_2 + c_5^1 W_3\right)$$

$$\left(a_0^2 + a_1^2 I_1 + a_2^2 I_2 + a_3^2 W_1 + a_4^2 W_2 + a_5^2 W_3\right) \cdot \left(b_0^2 + b_1^2 I_1 + b_2^2 I_2 + b_3^2 W_1 + b_4^2 W_2 + b_5^2 W_3\right) = \left(c_0^2 + c_1^2 I_1 + c_2^2 I_2 + c_3^2 W_1 + c_4^2 W_2 + c_5^2 W_3\right)$$

$$\left(a_0^3 + a_1^3 I_1 + a_2^3 I_2 + a_3^3 W_1 + a_4^3 W_2 + a_5^3 W_3\right) \cdot \left(b_0^3 + b_1^3 I_1 + b_2^3 I_2 + b_3^3 W_1 + b_4^3 W_2 + b_5^3 W_3\right) = \left(c_0^3 + c_1^3 I_1 + c_2^3 I_2 + c_3^3 W_1 + c_4^3 W_2 + c_5^3 W_3\right)$$

$$\left(a_0^4 + a_1^4 I_1 + a_2^4 I_2 + a_3^4 W_1 + a_4^4 W_2 + a_5^4 W_3\right) \cdot \left(b_0^4 + b_1^4 I_1 + b_2^4 I_2 + b_3^4 W_1 + b_4^4 W_2 + b_5^4 W_3\right) = \left(c_0^4 + c_1^4 I_1 + c_2^4 I_2 + c_3^4 W_1 + c_4^4 W_2 + c_5^4 W_3\right)$$

In what follows, we write L_{jubjub} for the associated language that consists of solutions to the R1CS.

To see that L_{jubjub} is equivalent to $L_{tiny.jj.1}$, let $(I_1, I_2; W_1, W_2, W_3)$ be a word in L_{jubjub}, then (I_1, I_2) is a word in $L_{tiny.jj.1}$, since the defining R1CS of L_{jubjub} implies that I_1 and I_2 satisfy the Edwards equation of the tiny jubjub curve. On the other hand, let (I_1, I_2) be a word in $L_{tiny.jj.1}$. Then $(I_1, I_2; I_1^2, I_2^2, 8 \cdot I_1^2 \cdot I_2^2)$ is a word in L_{jubjub} and both maps are inverses of each other.

Exercise 43. Consider the language $L_{tiny.jj\_zk}$ and define a rank-1 constraint relation with a decision function such that the associated language is equivalent to $L_{tiny.jj\_zk}$.

R1CS Satisfiability To understand how rank-1 constraint systems define formal languages, observe that every R1CS over a field $\mathbb{F}$ defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{R1CS} : \mathbb{F}^* \times \mathbb{F}^* \to \{true, false\} \; ; \; (I; W) \mapsto \begin{cases} true & (I; W) \text{ satisfies R1CS} \\ false & else \end{cases} \tag{6.7}$$

Every R1CS therefore defines a formal language. The grammar of this language is encoded in the constraints, words are solutions to the equations and a **statement** is a knowledge claim "Given instance I, there is a witness W such that $(I;W)$ is a solution to the rank-1 constraint system". A constructive proof to this claim is therefore an assignment of a field element to every witness variable, which is verified whenever the set of all instance and witness variables solves the R1CS.

Remark 2 (R1CS satisfiability). It should be noted that in our definition, every R1CS defines its own language. However, in more theoretical approaches, another language usually called **R1CS satisfiability** is often considered, which is useful when it comes to more abstract problems like expressiveness or the computational complexity of the class of **all** R1CS. From our perspective, the R1CS satisfiability language is obtained by the union of all R1CS languages that are in our definition. To be more precise, let the alphabet $\Sigma = \mathbb{F}$ be a field. Then

$$L_{R1CS\_SAT(\mathbb{F})} = \{(i;w) \in \Sigma^* \times \Sigma^* \mid \text{there is a R1CS } R \text{ such that } R(i;w) = true\}$$

Example 113 (3-Factorization). Consider the language $L_{3.fac\_zk}$ from example 106 and the R1CS defined in example ex:3-factorization-r1cs. As we have seen in ex:3-factorization-r1cs, solutions to the R1CS are in 1:1 correspondence with solutions to the decision function of $L_{3.fac\_zk}$. Both languages are therefore equivalent in the sense that there is a 1:1 correspondence between words in both languages.

To give an intuition of what constructive proofs in $L_{3.fac\_zk}$ look like, consider the instance $I_1 = 11$. To prove the statement "There exists a witness W such that $(I_1;W)$ is a word in $L_{3.fac\_zk}$" constructively, a proof has to provide assignments to all witness variables W_1, W_2, W_3 and W_4. Since the alphabet is $\mathbb{F}_{13}$, an example assignment is given by $W = (2,3,4,6)$ since $(I_1;W)$ satisfies the R1CS

$$W_1 \cdot W_2 = W_4 \qquad\qquad \# \, 2 \cdot 3 = 6$$
$$W_4 \cdot W_3 = I_1 \qquad\qquad \# \, 6 \cdot 4 = 11$$

A proper constructive proof is therefore given by $P = (2,3,4,6)$. Of course, P is not the only possible proof for this statement. Since factorization is not unique in a field in general, another constructive proof is given by $P' = (3,5,12,2)$.

Example 114 (The tiny jubjub curve). Consider the language L_{jubjub} from example 107 and its associated R1CS. To see how constructive proofs in L_{jubjub} look like, consider the instance $(I_1,I_2) = (11,6)$. To prove the statement "There exists a witness W such that $(I_1,I_2;W)$ is a word in L_{jubjub}" constructively, a proof has to provide assignments to all witness variables W_1, W_2 and W_3. Since the alphabet is $\mathbb{F}_{13}$, an example assignment is given by $W = (4,10,8)$ since $(I_1,I_2;W)$ satisfies the R1CS

$$I_1 \cdot I_1 = W_1 \qquad\qquad 11 \cdot 11 = 4$$
$$I_2 \cdot I_2 = W_2 \qquad\qquad 6 \cdot 6 = 10$$
$$(8 \cdot W_1) \cdot W_2 = W_3 \qquad\qquad (8 \cdot 4) \cdot 10 = 8$$
$$(12 \cdot W_2 + W_3 + 10 \cdot W_1 + 1) \cdot 1 = 0 \qquad\qquad 12 \cdot 10 + 8 + 10 \cdot 4 + 1 = 0$$

A proper constructive proof is therefore given by $P = (4,10,8)$, which shows that the instance $(11,6)$ is a point on the tiny jubjub curve.

Modularity As we discussed on page 121 XXX, it is often useful to construct complex state- `check reference` ments and their representing languages from simple ones. Rank-1 constraint systems are particularly useful for this, as the intersection of two R1CS over the same alphabet results in a new R1CS over that same alphabet.

To be more precise, let S_1 and S_2 be two R1CS over $\mathbb{F}$, then a new R1CS S_3 is obtained by the intersection $S_3 = S_1 \cap S_2$ of S_1 and S_2. In this context, intersection means that both the equations of S_1 **and** the equations of S_2 have to be satisfied in order to provide a solution for the system S_3.

As a consequence, developers are able to construct complex R1CS from simple ones and this modularity provides the theoretical foundation for many R1CS compilers, as we will see in XXX. `add reference`

6.2.2 Algebraic Circuits

As we have seen in the previous paragraphs, rank-1 constraint systems are quadratic equations such that solutions are knowledge proofs for the existence of words in associated languages. From the perspective of a proofer, it is therefore important to solve those equations efficiently.

However, in contrast to systems of linear equation, no general methods are known that solve systems of quadratic equations efficiently. Rank-1 constraint systems are therefore impractical from a proofers perspective and auxiliary information is needed that helps to compute solutions efficiently.

Methods which compute R1CS solutions are sometimes called **witness generator functions**. To provide a common example, we introduce another class of decision functions called **algebraic circuits**. As we will see, every algebraic circuit defines an associated R1CS and also provides an efficient way to compute solutions for that R1CS.

It can be shown that every space- and time-bounded computation is expressible as an algebraic circuit. Transforming high-level computer programs into those circuits is a process often called **flattening**.

To understand this in more detail, we will introduce our model for algebraic circuits and look at the concept of circuit execution and valid assignments. After that, we will show how to derive rank-1 constraint systems from circuits and how circuits are useful to compute solutions to their R1CS efficiently.

Algebraic circuit representation To see what algebraic circuits are, let $\mathbb{F}$ be a field. An algebraic circuit is then a directed acyclic (multi)graph that computes a polynomial function over $\mathbb{F}$. Nodes with only outgoing edges (source nodes) represent the variables and constants of the function and nodes with only incoming edges (sink nodes) represent the outcome of the function. All other nodes have exactly two incoming edges and represent the defining field operations **addition** as well as **multiplication**. Graph edges represent the flow of the computation along the nodes.

To be more precise, we call a directed acyclic multi-graph $C(\mathbb{F})$ an **algebraic circuit** over $\mathbb{F}$ in this book if the following conditions hold:

- The set of edges has a total order.

- Every source node has a label that represents either a variable or a constant from the field $\mathbb{F}$.

- Every sink node has exactly one incoming edge and a label that represents either a variable or a constant from the field $\mathbb{F}$.

- Every node that is neither a source nor a sink has exactly two incoming edges and a label from the set $\{+, *\}$ that represents either addition or multiplication in $\mathbb{F}$.

- All outgoing edges from a node have the same label.

- Outgoing edges from a node with a label that represents a variable have a label.

- Outgoing edges from a node with a label that represents multiplication have a label, if there is at least one labeled edge in both input path.

- All incoming edges to sink nodes have a label.

- If an edge has two labels S_i and S_j it gets a new label $S_i = S_j$.

- No other edge has a label.

- Incoming edges to sink nodes that are labeled with a constant $c \in \mathbb{F}$ are labeled with the same constant. Every other edge label is taken from the set $\{W, I\}$ and indexed compatible with the order of the edge set.

It should be noted that the details in the definitions of algebraic circuits vary between different sources. We use this definition as it is conceptually straightforward and well-suited for pen-and-paper computations.

To get a better intuition of our definition, let $C(\mathbb{F})$ be an algebraic circuit. Source nodes are the inputs to the circuit and either represent variables or constants. In a similar way, sink nodes represent termination points of the circuit and are either output variables or constants. Constant sink nodes enforce computational outputs to take on certain values.

Nodes that are neither source nodes nor sink nodes are called **arithmetic gates**. Arithmetic gates that are decorated with the "+"-label are called **addition-gates** and arithmetic gates that are decorated with the "·"-label are called **multiplication-gates**. Every arithmetic gate has exactly two inputs, represented by the two incoming edges.

Since the set of edges is ordered, we can write it as $\{E_1, E_2, \ldots, E_n\}$ for some $n \in \mathbb{N}$ and we use those indices to index the edge labels, too. Edge labels are therefore either constants or symbols like I_j, W_j or S_j, where j is an index compatible with the edge order. Labels I_j represent instance variables, labels W_j witness variables. Labels on the outgoing edges of input variables constrain the associated variable to that edge. Every other edge defines a constraining equation in the associated R1CS. we will explain this in more detail in XXX. [add reference]

Notation and Symbols 10. In synthesizing algebraic circuits, assigning instance I_j or witness W_j labels to appropriate edges is often the final step. It is therefore convenient to not distinguish these two types of edges in previous steps. To account for that, we often simply write S_j for an edge label, indicating that the private/public property of the label is unspecified and it might represent an instance or a witness label.

Example 115 (Generalized factorization SNARK). To give a simple example of an algebraic circuit, consider our 3-factorization problem from example 106 again. To express the problem [check reference] in the algebraic circuit model, consider the following function

$$f_{3.fac} : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \to \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto x_1 \cdot x_2 \cdot x_3$$

Using this function, we can describe the zero-knowledge 3-factorization problem from 106, [check reference] in the following way: Given instance $I_1 \in \mathbb{F}_{13}$, a valid witness is a preimage of $f_{3.fac}$ at

the point I_1, i.e., a valid witness consists of three values W_1, W_2 and W_3 from $\mathbb{F}_{13}$ such that $f_{3.fac}(W_1,W_2,W_3) = I_1$.

To see how this function can be transformed into an algebraic circuit over $\mathbb{F}_{13}$, it is a common first step to introduce brackets into the function's definition and then write the operations as binary operators, in order to highlight how exactly every field operation acts on its two inputs. Due to the associativity laws in a field, we have several choices. We choose

$$
\begin{aligned}
f_{3.fac}(x_1,x_2,x_3) &= x_1 \cdot x_2 \cdot x_3 && \text{\# bracket choice} \\
&= (x_1 \cdot x_2) \cdot x_3 && \text{\# operator notation} \\
&= MUL(MUL(x_1,x_2),x_3)
\end{aligned}
$$

Using this expression, we can write an associated algebraic circuit by first constraining the variables to edge labels $W_1 = x_1$, $W_2 = x_2$ and $W_3 = x_3$ as well as $I_1 = f_{3.fac}(x_1,x_2,x_3)$, taking the distinction between private and public inputs into account. We then rewrite the operator representation of $f_{3.fac}$ into circuit nodes and get the following:
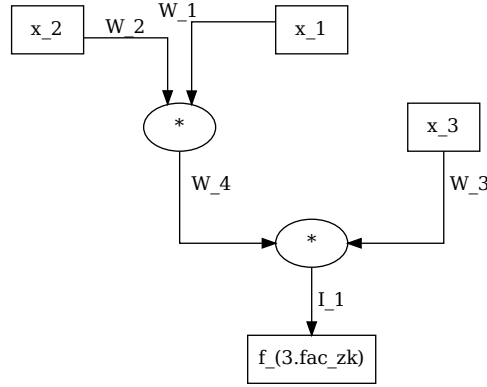
In this case, the directed acyclic multi-graph is a binary tree with three leaves (the source nodes) labeled by x_1, x_2 and x_3, one root (the single sink node) labeled by $f(x_1,x_2,x_3)$ and two internal nodes, which are labeled as multiplication gates.

The order we used to label the edges is chosen to make the edge labeling consistent with the choice of W_4 as defined in example XXX. This order can be obtained by adepth-first right-to-left-first traversal algorithm. [add reference]

Example 116. To give a more realistic example of an algebraic circuit, look at the defining equation XXX of the tiny-jubjub curve again. A pair of field elements $(x,y) \in \mathbb{F}_{13}^2$ is a curve point, precisely if [add reference]

$$
3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2
$$

To understand how one might transform this identity into an algebraic circuit, we first rewrite this equation by shifting all terms to the right. We get:

$$
\begin{aligned}
3 \cdot x^2 + y^2 &= 1 + 8 \cdot x^2 \cdot y^2 && \Leftrightarrow \\
0 &= 1 + 8 \cdot x^2 \cdot y^2 - 3 \cdot x^2 - y^2 && \Leftrightarrow \\
0 &= 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2
\end{aligned}
$$

Then we use this expression to define a function such that all points of the tiny-jubjub curve are characterized as the function preimages at 0.

$$f_{tiny-jj} : \mathbb{F}_{13} \times \mathbb{F}_{13} \to \mathbb{F}_{13} \; ; \; (x,y) \mapsto 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12 \cdot y^2$$

Every pair of points $(x,y) \in \mathbb{F}_{13}^2$ with $f_{tiny-jj}(x,y) = 0$ is a point on the tiny-jubjub curve, and there are no other curve points. The preimage $f_{tiny-jj}^{-1}(0)$ is therefore a complete description of the tiny-jubjub curve.

We can transform this function into an algebraic circuit over $\mathbb{F}_{13}$. We first introduce brackets into potentially ambiguous expressions and then rewrite the function in terms of binary operators. We get

$$\begin{aligned} f_{tiny-jj}(x,y) &= 1 + 8 \cdot x^2 \cdot y^2 + 10 \cdot x^2 + 12y^2 & \Leftrightarrow \\ &= ((8 \cdot ((x \cdot x) \cdot (y \cdot y))) + (1 + 10 \cdot (x \cdot x))) + (12 \cdot (y \cdot y)) & \Leftrightarrow \\ &= ADD(ADD(MUL(8,MUL(MUL(x,x),MUL(y,y))),ADD(1,MUL(10,MUL(x,x)))),MUL(12,MUL(y,y))) \end{aligned}$$

Since we haven't decided which part of the computation should be public and which part should be private, we use the unspecified symbol S to represent edge labels. Constraining all variables to edge labels $S_1 = x$, $S_2 = y$ and $S_6 = f_{tiny-jj}$, we get the following circuit, representing the function $f_{tiny-jj}$, by inductively replacing binary operators with their associated arithmetic gates:
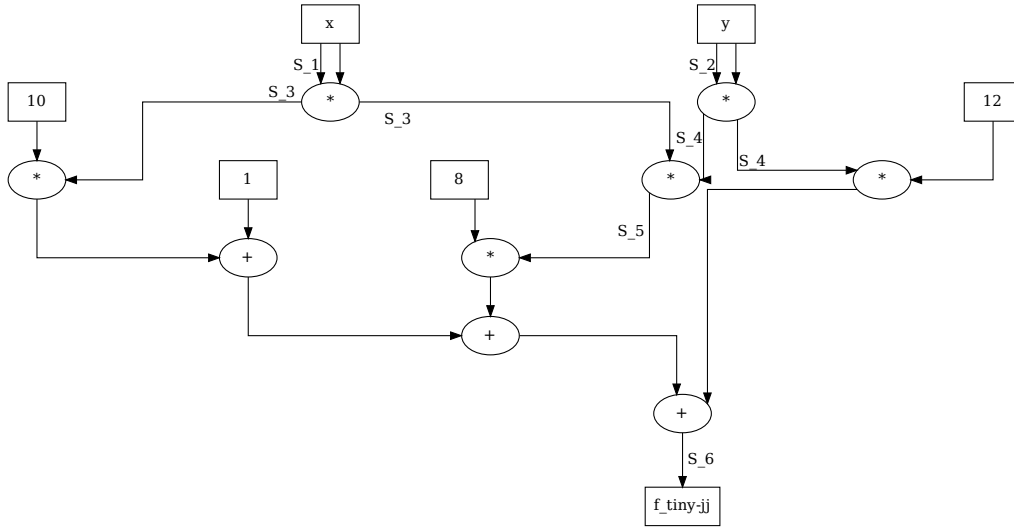
This circuit is not a graph, but a multigraph, since there is more than one edge between some of the nodes.

In the process of designing of circuits from functions, it should be noted that circuit representations are not unique in general. In case of the function $f_{tiny-jj}$, the circuit shape is dependent on our choice of bracketing in XXX. An alernative design i,s for example, given by the following circuit, which occurs when the bracketed expression $8 \cdot ((x \cdot x) \cdot (y \cdot y))$ is replaced by the expression $(x \cdot x) \cdot (8 \cdot (y \cdot y))$.

add refer- ence
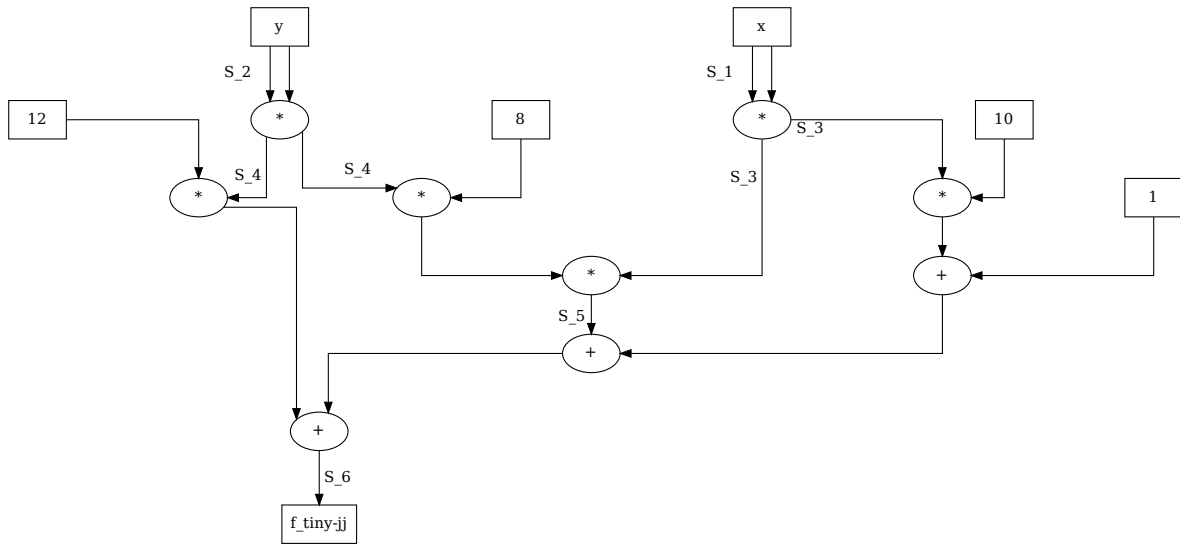
Of course, both circuits represent the same function, due to the associativity and commutativity laws that hold true in any field.

With a circuit that represents the function $f_{tiny-jj}$, we can now proceed to derive a circuit that constrains arbitrary pairs (x, y) of field elements to be points on the tiny-jubjub curve. To do so, we have to constrain the output to be zero, that is, we have to constrain $S_6 = 0$. To indicate this in the circuit, we replace the output variable by the constant 0 and constrain the related edge label accordingly. We get
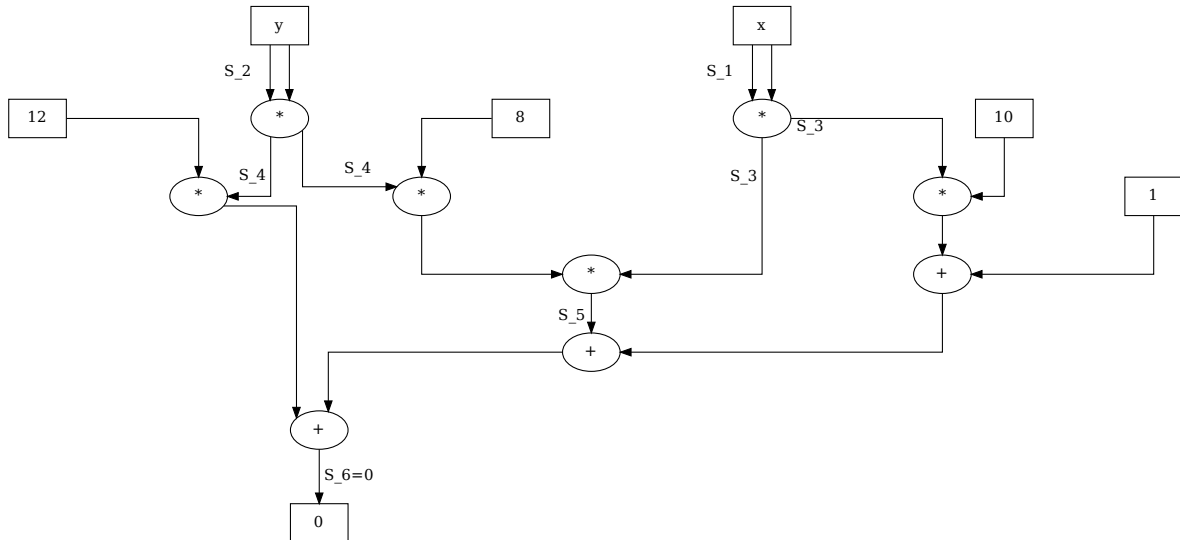
The previous circuit enforces input values assigned to the labels S_1 and S_2 to be points on the tiny jubjub curve. However, it does not specify which labels are considered public and which are considered private. The following circuit defines the inputs to be public, while all other labels are private:
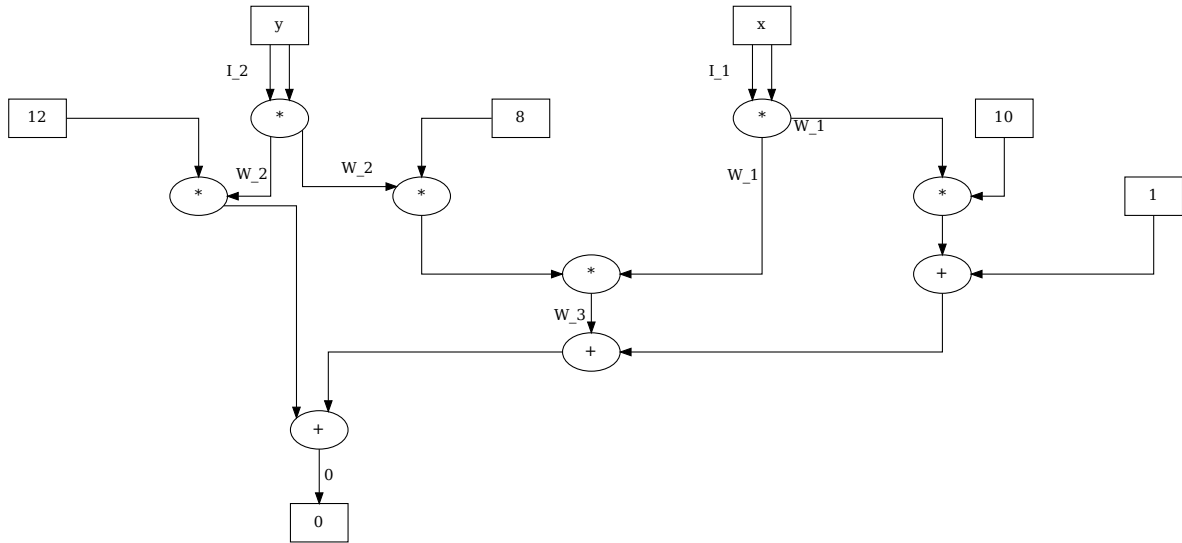
It can be shown that every space- and time-bounded computation can be transformed into an algebraic circuit. We call any process that transforms a bounded computation into a circuit **flattening**.

We already said this in this chapter

Circuit Execution   Algebraic circuits are directed, acyclic multi-graphs, where nodes represent variables, constants, or addition and multiplication gates. In particular, every algebraic circuit with n input nodes decorated with variable symbols and m output nodes decorated with variables can be seen a function that transforms an input tuple $(x_1, \ldots, x_n)$ from $\mathbb{F}^n$ into an output tuple $(f_1, \ldots, f_m)$ from $\mathbb{F}^m$. The transformation is done by sending values associated to nodes along their outgoing edges to other nodes. If those nodes are gates, then the values are transformed according to the gate label and the process is repeated along all edges until a sink node is reached. We call this computation **circuit execution**.

When executing a circuit, it is possible to not only compute the output values of the circuit but to derive field elements for all edges, and, in particular, for all edge labels in the circuit. The result is a tuple $(S_1, S_2, \ldots, S_n)$ of field elements associated to all labeled edges, which we call a **valid assignment** to the circuit. In contrast, any assignment $(S'_1, S'_2, \ldots, S'_n)$ of field elements to edge labels that can not arise from circuit execution is called an **invalid assignment**.

Valid assignments can be interpreted as **proofs for proper circuit execution** because they keep a record of the computational result as well as intermediate computational steps.

Example 117 (3-factorization). Consider the 3-factorization problem from example 106 and its representation as an algebraic circuit from XXX. We know that the set of edge labels is given by $S := \{I_1; W_1, W_2, W_3, W_4\}$.

check reference

add reference

To understand how this circuit is executed, consider the variables $x_1 = 2$, $x_2 = 3$ as well as $x_3 = 4$. Following all edges in the graph, we get the assignments $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$. Then the assignments of W_1 and W_2 enter a multiplication gate and the output of the gate is $2 \cdot 3 = 6$, which we assign to W_4, i.e. $W_4 = 6$. The values W_4 and W_3 then enter the second multiplication gate and the output of the gate is $6 \cdot 4 = 11$, which we assign to I_1, i.e. $I_1 = 11$.

A valid assignment to the 3-factorization circuit $C_{3.fac}(\mathbb{F}_{13})$ is therefore given by the set $S_{valid} := \{11; 2, 3, 4, 6\}$. We can picture this assignment in the circuit as follows:
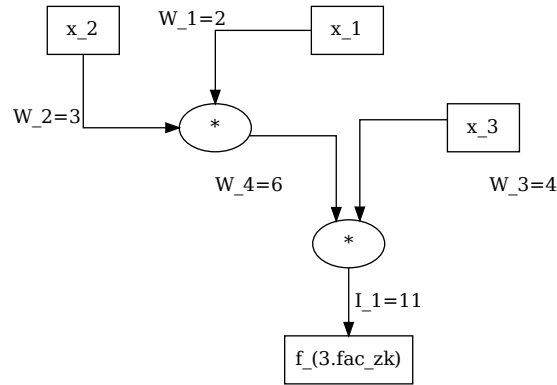
To see what an invalid assignment looks like, consider the assignment $S_{err} := \{8; 2, 3, 4, 7\}$. In this assignment, the input values are the same as in the previous case. The associated circuit is:
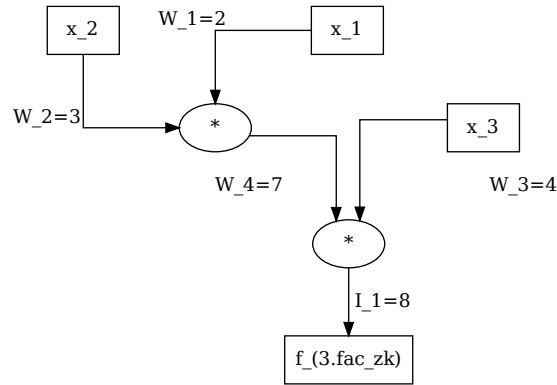
This assignment is invalid, as the assignments of I_1 and W_4 cannot be obtained by executing the circuit.

Example 118. To compute a more realistic algebraic circuit execution, consider the defining circuit $C_{tiny-jj}(\mathbb{F}_{13})$ from example 114 again. We already know from the way this circuit is constructed that any valid assignment with $S_1 = x$, $S_2 = y$ and $S_6 = 0$ will ensure that the pair (x, y) is a point on the tiny jubjub curve XXX in its Edwards representation.

From example 114 we know that the pair $(11, 6)$ is a proper point on the tiny-jubjub curve and we use this point as input to a circuit execution. We get:

check reference

add reference

check reference
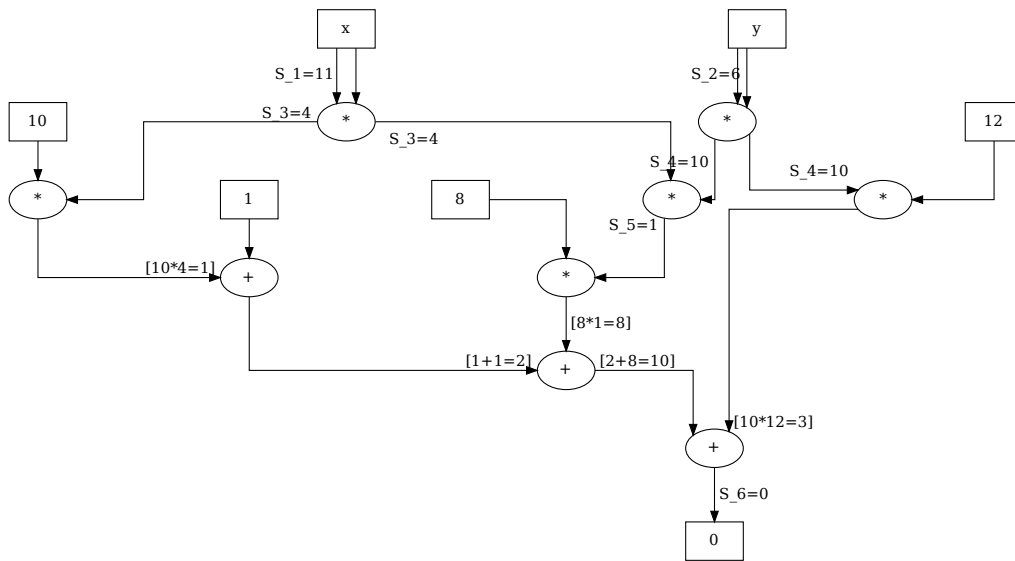
Executing the circuit, we indeed compute $S_6 = 0$ as expected, which proves that $(11, 6)$ is a point on the tiny-jubjub curve in its Edwards representation. A valid assignment of $C_{tiny-jj}(\mathbb{F}_{13})$ is therefore given by

$$S_{tiny-jj} = \{S_1, S_2, S_3, S_4, S_5, S_6\} = \{11, 6, 4, 10, 1, 0\}$$

Circuit Satisfiability   To understand how algebraic circuits give rise to formal languages, observe that every algebraic circuit $C(\mathbb{F})$ over a fields $\mathbb{F}$ defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{C(\mathbb{F})} : \mathbb{F}^* \times \mathbb{F}^* \to \{true, false\} \; ; \; (I; W) \mapsto \begin{cases} true & (I; W) \text{is valid assignment to } C(\mathbb{F}) \\ false & else \end{cases} \quad (6.8)$$

Every algebraic circuit therefore defines a formal language. The grammar of this language is encoded in the shape of the circuit, words are assignments to edge labels that are derived from circuit execution, and **statements** are knowledge claims "Given instance I, there is a witness W such that $(I; W)$ is a valid assignment to the circuit". A constructive proof to this claim is therefore an assignment of a field element to every witness variable, which is verified by executing the circuit to see if the assignment of the execution meets the assignment of the proof.

   In the context of zero-knowledge proof systems, executing circuits is also often called **witness generation**, since in applications the instance part is usually public, while its the task of a proofer to compute the witness part.

Remark 3 (Circuit satisfiability). It should be noted that, in our definition, every circuit defines its own language. However, in more theoretical approaches another language usually called **circuit satisfiability** is often considered, which is useful when it comes to more abstract problems like expressiveness, or computational complexity of the class of **all** algebraic circuits over a given field. From our perspective the circuit satisfiability language is obtained by union of all circuit languages that are in our definition. To be more precise, let the alphabet $\Sigma = \mathbb{F}$ be a field. Then

$$L_{CIRCUIT\_SAT(\mathbb{F})} = \{(i; w) \in \Sigma^* \times \Sigma^* \mid \text{there is a circuit } C(\mathbb{F}) \text{ such that } (i; w) \text{ is valid assignment}\}$$

Example 119 (3-Factorization). Consider the circuit $C_{3.fac}$ from example XXX again. We call the associated language $L_{3.fac\_circ}$.

To understand how a constructive proof of a statement in $L_{3.fac\_circ}$ looks like, consider the instance $I_1 = 11$. To provide a proof for the statement "There exist a witness W such that $(I_1; W)$ is a word in $L_{3.fac\_circ}$" a proof therefore has to consists of proper values for the variables W_1, W_2, W_3 and W_4. Any proofer therefore has to find input values for W_1, W_2 and W_3 and then execute the circuit to compute W_4 under the assumption $I_1 = 11$.

Example XXX implies that $(2, 3, 4, 6)$ is a proper constructive proof and in order to verify the proof a verifier needs to execute the circuit with instance $I_1 = 11$ and inputs $W_1 = 2$, $W_2 = 3$ and $W_3 = 4$ to decide whether the proof is a valid assignment or not.

Associated Constraint Systems   As we have seen in XXX, rank-1 constraint systems define a way to represent statements in terms of a system of quadratic equations over finite fields, suitable for pairing-based zero-knowledge proof systems. However, those equations provide no practical way for a proofer to actually compute a solution. On the other hand, algebraic circuits can be executed in order to derive valid assignments efficiently.

In this paragraph, we show how to transform any algebraic circuit into a rank-1 constraint system such that valid circuit assignments are in 1:1 correspondence with solutions to the associated R1CS.

To see this, let $C(\mathbb{F})$ be an algebraic circuit over a finite field $\mathbb{F}$, with a set of edge labels $\{S_1, S_2, \ldots, S_n\}$. Then one of the following steps is executed for every edge label S_j from that set:

- If the edge label S_j is an outgoing edge of a multiplication gate, the R1CS gets a new quadratic constraint

$$(\text{left input}) \cdot (\text{right input}) = S_j \tag{6.9}$$

  where (left input) respectively (right input) is the output from the symbolic execution of the subgraph that consists of the left respectively right input edge of this gate, and all edges and nodes that have this edge in their path, starting with constant inputs or labeled outgoing edges of other nodes.

- If the edge label S_j is an outgoing edge of an addition gate, the R1CS gets a new quadratic constraint

$$(\text{left input} + \text{right input}) \cdot 1 = S_j \tag{6.10}$$

  where (left input) respectively (right input) is the output from the symbolic execution of the subgraph that consists of the left respectively right input edge of this gate and all edges and nodes that have this edge in their path, starting with constant inputs or labeled outgoing edges of other nodes.

- No other edge label adds a constraint to the system.

The result of this method is a rank-1 constraint system, and in this sense, every algebraic circuit $C(\mathbb{F})$ generates a R1CS R, which we call the **associated R1CS** of the circuit. It can be shown that a tuple of field elements $(S_1, S_2, \ldots, S_n)$ is a valid assignment to a circuit if and only if the same tuple is a solution to the associated R1CS. Circuit executions therefore compute solutions to rank-1 constraints systems efficiently.

To understand the contribution of algebraic gates to the number of constraints, note that by definition multiplication gates have labels on their outgoing edges if and only if there is at least one labeled edge in both input paths, or if the outgoing edge is an input to a sink node. This

implies that multiplication with a constant is essentially free in the sense that it doesn't add a new constraint to the system, as long as that multiplication gate is not am input to an output node.

Moreover, addition gates have labels on their outgoing edges if and only if they are inputs to sink nodes. This implies that addition is essentially free in the sense that it doesn't add a new constraint to the system, as long as that addition gate is not an input to an output node.

Example 120 (3-factorization). Consider our 3-factorization problem from example XXX and the associated circuit $C_{3.fac}(\mathbb{F}_{13})$. Out task is to transform this circuit into an equivalent rank-1 constraint system. `add reference`
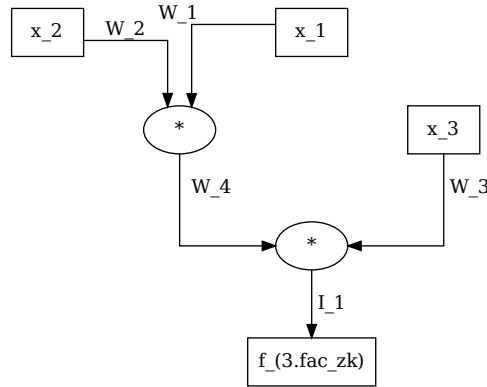
We start with an empty R1CS, and, in order to generate all constraints, we have to iterate over the set of edge labels $\{I_1; W_1, W_2, W_3, W_4\}$.

Starting with the edge label I_1, we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

$$(\text{left input}) \cdot (\text{right input}) = I_1 \qquad \Leftrightarrow$$
$$W_4 \cdot W_3 = I_1$$

Next, we consider the edge label W_1 and, since, it's not an outgoing edge of a multiplication or addition label, we don't add a constraint to the system. The same holds true for the labels W_2 and W_3.

For edge label W_4 , we see that it is an outgoing edge of a multiplication gate, and, since both input edges are labeled, we have to add the following constraint to the system:

$$(\text{left input}) \cdot (\text{right input}) = W_4 \qquad \Leftrightarrow$$
$$W_2 \cdot W_1 = W_4$$

Since there are no more labeled edges, all constraints are generated, and we have to combine them to get the associated R1CS of $C_{3.fac}(\mathbb{F}_{13})$:

$$W_4 \cdot W_3 = I_1$$
$$W_2 \cdot W_1 = W_4$$

This system is equivalent to the R1CS we derived in example 111. The languages $L_{3.fac\_zk}$ and $L_{3.fac\_circ}$ are therefore equivalent and both the circuit as well as the R1CS are just two different ways of expressing the same language. `check reference`

Example 121. To consider a more general transformation, we consider the tiny-jubjub circuit from example 114 again. A proper circuit is given by

check
reference
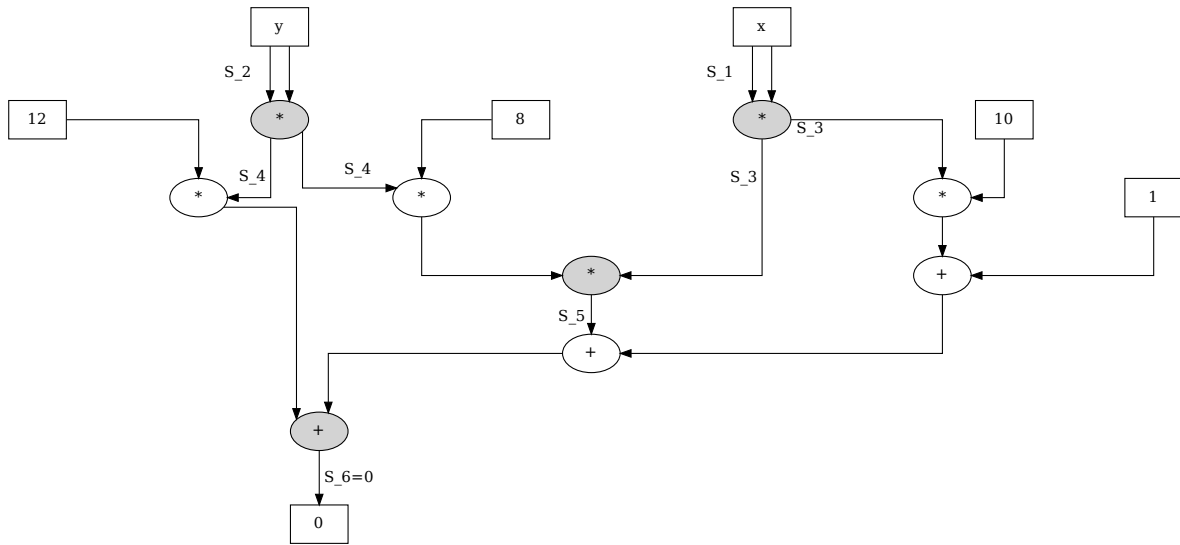
To compute the number of constraints, observe that we have 3 multiplication gates that have labels on their outgoing edges and 1 addition gate that has a label on its outgoing edge. We therefore have to compute 4 quadratic constraints.

In order to derive the associated R1CS, we have start with an empty R1CS and then iterate over the set $\{S_1, S_2, S_3, S_4, S_5, S_6 = 0\}$ of all edge labels, in order to generate the constraints.

Considering edge label S_1, we see that the associated edges are not outgoing edges of any algebraic gate, and we therefore have to add no new constraint to the system. The same holds true for edge label S_2. Looking at edge label S_3, we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:
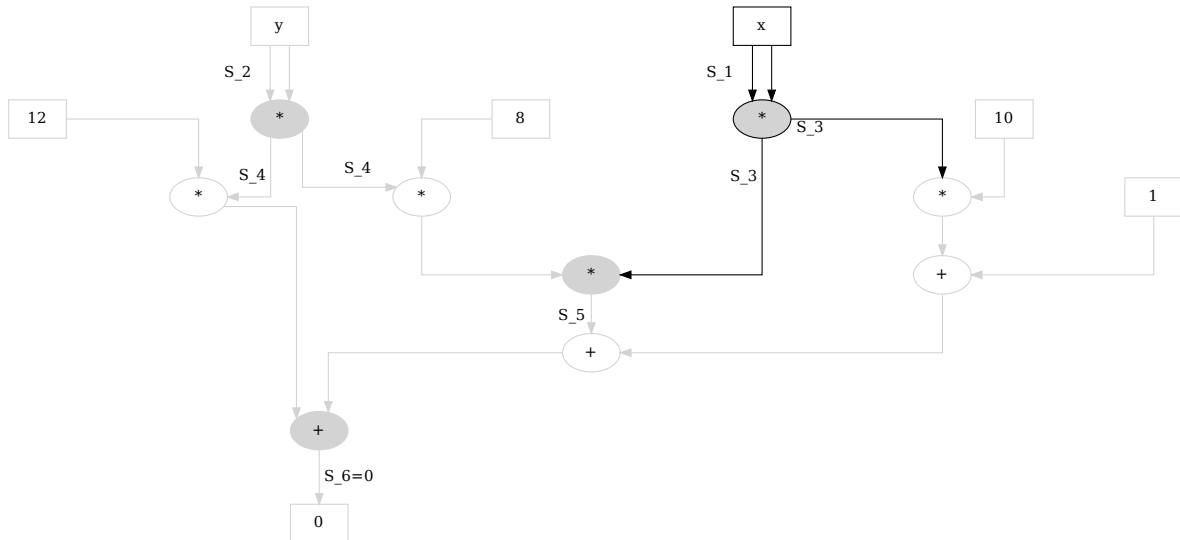
Both the left and the right input to this multiplication gate are labeled by S_1. We therefore have to add the following constraint to the system:

$$S_1 \cdot S_1 = S_3$$

Looking at edge label S_4, we see that the associated edges are outgoing edges of a multiplication gate and that the associated subgraph is given by:
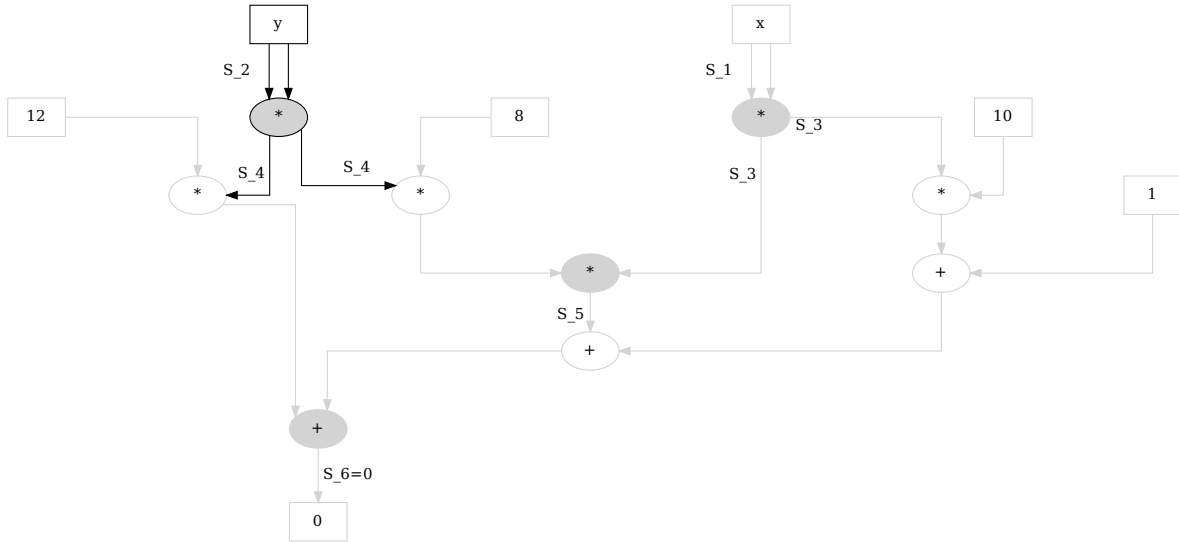
Both the left and the right input to this multiplication gate are labeled by S_2 and we therefore have to add the following constraint to the system:

$$S_2 \cdot S_2 = S_4$$

Edge label S_5 is more interesting. To see if it implies a constraint, we have to construct the associated subgraph first, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get

137
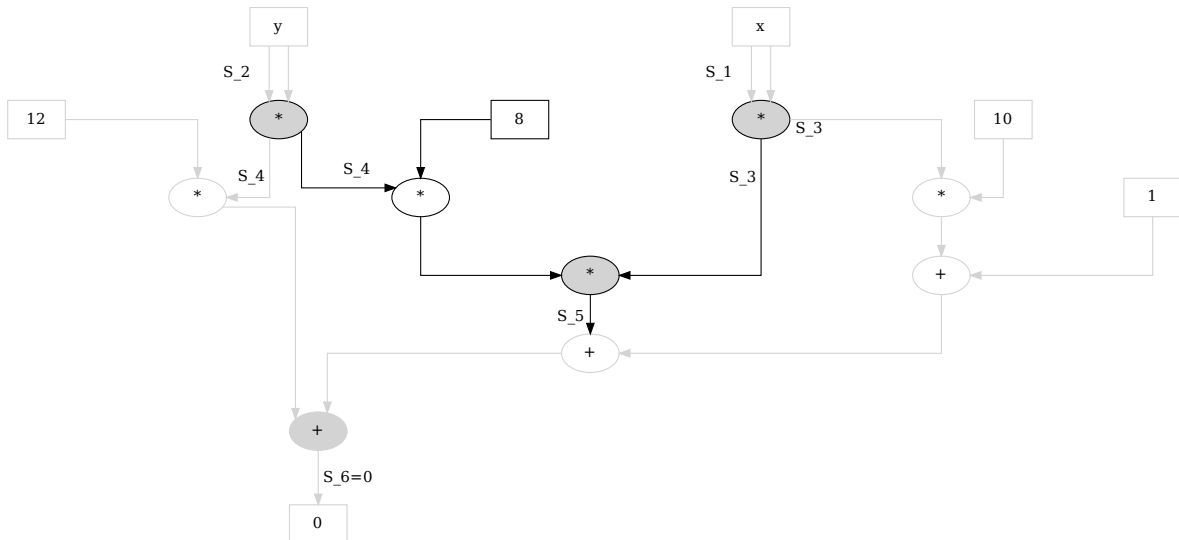
The right input to the associated multiplication gate is given by the labeled edge S_3. However, the left input is not a labeled edge, but has a labeled edge in one of its path. This implies that we have to add a constraint to the system. To compute the left factor of that constraint, we have to compute the output of subgraph associated to the left edge, which is $8 \cdot W_2$. This gives the constraint

$$(S_4 \cdot 8) \cdot S_3 = S_5$$

The last edge label is the constant $S_6 = 0$. To see if it implies a constraint, we have to construct the associated subgraph, which consists of all edges and all nodes in all path starting either at a constant input or a labeled edge. We get
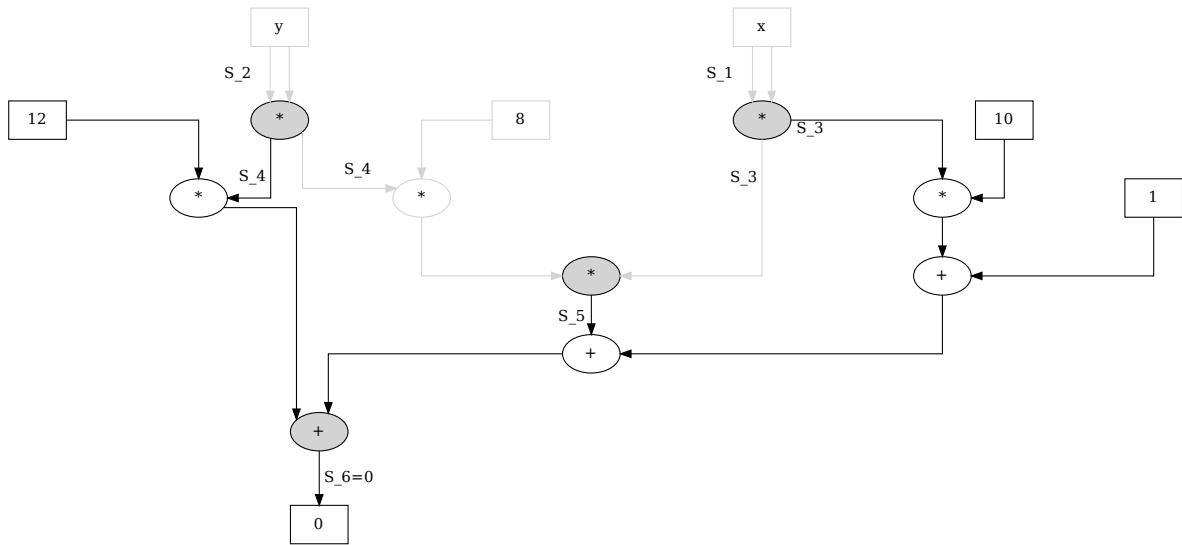
Both the left and the right input are unlabeled, but have a labeled edges in their path. This implies that we have to add a constraint to the system. Since the gate is an addition gate, the right factor in the quadratic constraint is always 1 and the left factor is computed by symbolically executing all inputs to all gates in sub-circuit. We get

$$(12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 = 0$$

Since there are no more labeled outgoing edges, we are done deriving the constraints. Combining all constraints together, we get the following R1CS:

$$S_1 \cdot S_1 = S_3$$
$$S_2 \cdot S_2 = S_4$$
$$(S_4 \cdot 8) \cdot S_3 = S_5$$
$$(12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 = 0$$

which is equivalent to the R1CS we derived in example 114. The languages $L_{3.fac\_zk}$ and $L_{3.fac\_circ}$ are therefore equivalent and both the circuit as well as the R1CS are just two different ways to express the same language.

6.2.3 Quadratic Arithmetic Programs

We have introduced algebraic circuits and their associated rank-1 constraints systems as two particular models able to represent space- and time-bounded computation. Both models define formal languages, and associated membership as well as knowledge claims can be constructively proofed by executing the circuit in order to compute solutions to its associated R1CS.

One reason why those systems are useful in the context of succinct zero-knowledge proof systems is because any R1CS can be transformed into another computational model called **quadratic arithmetic programs** (QAP), which serve as the basis for some of the most efficient succinct non-interactive zero-knowledge proof generators that currently exist.

As we will see, proving statements for languages that have checking relations defined by quadratic arithmetic programs can be achieved by providing certain polynomials, and those proofs can be verified by checking a particular divisibility property.

QAP representation   To understand what quadratic arithmetic programs are in detail, let $\mathbb{F}$ be a field and R a rank-1 constraints system over $\mathbb{F}$ such that the number of non-zero elements in $\mathbb{F}$ is strictly larger then the number k of constraints in R. Moreover, let a_j^i, b_j^i and $c_j^i \in \mathbb{F}$ for every index $0 \leq j \leq n+m$ and $1 \leq i \leq k$, be the defining constants of the R1CS and $m_1, \ldots, m_k$ be arbitrary, invertible and distinct elements from $\mathbb{F}$.

Then a **quadratic arithmetic program** [QAP] of the R1CS is the following set of polynomials over $\mathbb{F}$

$$QAP(R) = \left\{ T \in \mathbb{F}[x], \left\{ A_j, B_j, C_j \in \mathbb{F}[x] \right\}_{h=0}^{n+m} \right\} \tag{6.11}$$

where $T(x) := \Pi_{l=1}^k (x - m_l)$ is a polynomial of degree k, called the **target polynomial** of the QAP and A_j, B_j as well as C_j are the unique degree $k-1$ polynomials defined by the equations

$$A_j(m_i) = a_j^i \quad B_j(m_i) = b_j^i \quad C_j(m_i) = C_j^i \quad j = 1, \ldots, n+m+1, i = 1, \ldots, k \tag{6.12}$$

Given some rank-1 constraint system, an associated quadratic arithmetic program is therefore nothing but a set of polynomials, computed from the constants in the R1CS. To see that the polynomials A_j, B_j and C_j are uniquely defined by the equations in XXX, recall that a a polynomial of degree $k-1$ is completely determined on k evaluation points and the equation XXX precisely determines those k evaluation points.

Since we only consider polynomials over fields, Lagrange's interpolation method from 3.31 in chapter 3 can be used to derive the polynomials A_j, B_j and C_j from their defining equations XXX. A practical method to compute a QAP from a given R1CS therefore consists of two steps. If the R1CS consists of k constraints, first choose k invertible and mutually different points from the underlaying field. Every choice defines a different QAP for the same R1CS. Then use Lagrange's method and equation XXX to compute the polynomials A_j, B_j and C_j for every $1 \leq j \leq k$.

Example 122 (Generalized factorization SNARK). To provide a better intuition of quadratic arithmetic programs and how they are computed from their associated rank-1 constraint systems, consider the language $L_{3.fac\_zk}$ from example 106 and its associated R1CS

$$\begin{aligned} W_1 \cdot W_2 &= W_4 & &\text{constraint 1} \\ W_4 \cdot W_3 &= I_1 & &\text{constraint 2} \end{aligned}$$

In this example we want to transform this R1CS into an associated QAP. In a first step, we have to compute the defining constants a_j^i, b_j^i and c_j^i of the R1CS. According to XXX, we have

139

$$a_0^1 = 0 \quad a_1^1 = 0 \quad a_2^1 = 1 \quad a_3^1 = 0 \quad a_4^1 = 0 \quad a_5^1 = 0$$
$$a_0^2 = 0 \quad a_1^2 = 0 \quad a_2^2 = 0 \quad a_3^2 = 0 \quad a_4^2 = 0 \quad a_5^2 = 1$$

$$b_0^1 = 0 \quad b_1^1 = 0 \quad b_2^1 = 0 \quad b_3^1 = 1 \quad b_4^1 = 0 \quad b_5^1 = 0$$
$$b_0^2 = 0 \quad b_1^2 = 0 \quad b_2^2 = 0 \quad b_3^2 = 0 \quad b_4^2 = 1 \quad b_5^2 = 0$$

$$c_0^1 = 0 \quad c_1^1 = 0 \quad c_2^1 = 0 \quad c_3^1 = 0 \quad c_4^1 = 0 \quad c_5^1 = 1$$
$$c_0^2 = 0 \quad c_1^2 = 1 \quad c_2^2 = 0 \quad c_3^2 = 0 \quad c_4^2 = 0 \quad c_5^2 = 0$$

Since the R1CS is defined over the field $\mathbb{F}_{13}$ and has two constraining equations, we need to choose two arbitrary but distinct elements m_1 and m_2 from $\mathbb{F}_{13}$. We choose $m_1 = 5$, and $m_2 = 7$ and with this choice we get the target polynomial

$$
\begin{aligned}
T(x) &= (x - m_1)(x - m_2) && \text{\# Definition of T} \\
&= (x - 5)(x - 7) && \text{\# Insert our choice} \\
&= (x + 8)(x + 6) && \text{\# Negatives in } \mathbb{F}_{13} \\
&= x^2 + x + 9 && \text{\# expand}
\end{aligned}
$$

Then we have to compute the polynomials A_j, B_j and C_j by their defining equation from the R1CS coefficients. Since the R1CS has two constraining equations, those polynomials are of degree 1 and they are defined by their evaluation at the point $m_1 = 5$ and the point $m_2 = 7$.

At point m_1, each polynomial A_j is defined to be a_j^1 and at point m_2, each polynomial A_j is defined to be a_j^2. The same holds true for the polynomials B_j as well as C_j. Writing all these equations now, we get:

$$A_0(5) = 0, \quad A_1(5) = 0, \quad A_2(5) = 1, \quad A_3(5) = 0, \quad A_4(5) = 0, \quad A_5(5) = 0$$
$$A_0(7) = 0, \quad A_1(7) = 0, \quad A_2(7) = 0, \quad A_3(7) = 0, \quad A_4(7) = 0, \quad A_5(7) = 1$$

$$B_0(5) = 0, \quad B_1(5) = 0, \quad B_2(5) = 0, \quad B_3(5) = 1, \quad B_4(5) = 0, \quad B_5(5) = 0$$
$$B_0(7) = 0, \quad B_1(7) = 0, \quad B_2(7) = 0, \quad B_3(7) = 0, \quad B_4(7) = 1, \quad B_5(7) = 0$$

$$C_0(5) = 0, \quad C_1(5) = 0, \quad C_2(5) = 0, \quad C_3(5) = 0, \quad C_4(5) = 0, \quad C_5(5) = 1$$
$$C_0(7) = 0, \quad C_1(7) = 1, \quad C_2(7) = 0, \quad C_3(7) = 0, \quad C_4(7) = 0, \quad C_5(7) = 0$$

Lagrange's interpolation implies that a polynomial of degree k, that is, that zero on $k + 1$ points has to be the zero polynomial. Since our polynomials are of degree 1 and determined on 2 points, we therefore know that the only non-zero polynomials in our QAP are A_2, A_5, B_3, B_4, C_1 and C_5, and that we can use Lagrange's interpolation to compute them.

To compute A_2 we note that the set S in our version of Lagrange's method is given by $S = \{(x_0, y_0), (x_1, y_1)\} = \{(5, 1), (7, 0)\}$. Using this set we get:

$$
\begin{aligned}
A_2(x) &= y_0 \cdot l_0 + y_1 \cdot l_1 \\
&= y_0 \cdot \left(\frac{x - x_1}{x_0 - x_1}\right) + y_1 \cdot \left(\frac{x - x_0}{x_1 - x_0}\right) = 1 \cdot \left(\frac{x - 7}{5 - 7}\right) + 0 \cdot \left(\frac{x - 5}{7 - 5}\right) \\
&= \frac{x - 7}{-2} = \frac{x - 7}{11} && \text{\# } 11^{-1} = 6 \\
&= 6(x - 7) = 6x + 10 && \text{\# } -7 = 6 \text{ and } 6 \cdot 6 = 10
\end{aligned}
$$

To compute A_5, we note that the set S in our version of Lagrange's method is given by $S = \{(x_0, y_0), (x_1, y_1)\} = \{(5, 0), (7, 1)\}$. Using this set we get:

$$A_5(x) = y_0 \cdot l_0 + y_1 \cdot l_1$$

$$= y_0 \cdot (\frac{x - x_1}{x_0 - x_1}) + y_1 \cdot (\frac{x - x_0}{x_1 - x_0}) = 0 \cdot (\frac{x - 7}{5 - 7}) + 1 \cdot (\frac{x - 5}{7 - 5})$$

$$= \frac{x - 5}{2} \qquad\qquad\qquad\qquad\qquad\qquad\qquad \# \, 2^{-1} = 7$$

$$= 7(x - 5) = 7x + 4 \qquad\qquad\qquad\qquad \# -5 = 8 \text{ and } 7 \cdot 8 = 4$$

Using Lagrange's interpolation, we can deduce that $A_2 = B_3 = C_5$ as well as $A_5 = B_4 = C_1$, since they are polynomials of degree 1 that evaluate to same values on 2 points. Using this, we get the following set of polynomials

$A_0(x) = 0$	$B_0(x) = 0$	$C_0(x) = 0$
$A_1(x) = 0$	$B_1(x) = 0$	$C_1(x) = 7x + 4$
$A_2(x) = 6x + 10$	$B_2(x) = 0$	$C_2(x) = 0$
$A_3(x) = 0$	$B_3(x) = 6x + 10$	$C_3(x) = 0$
$A_4(x) = 0$	$B_4(x) = 7x + 4$	$C_4(x) = 0$
$A_5(x) = 7x + 4$	$B_5(x) = 0$	$C_5(x) = 6x + 10$

We can invoke Sage to verify our computation. In sage every polynomial ring has a function `lagrange_polynomial` that takes the defining points as inputs and the associated Lagrange polynomial as output.

```
sage: F13 = GF(13)                                              1
sage: F13t.<t> = F13[]                                          2
sage: T = F13t((t-5)*(t-7))                                     3
sage: A2 = F13t.lagrange_polynomial([(5,1),(7,0)])             4
sage: A5 = F13t.lagrange_polynomial([(5,0),(7,1)])             5
sage: T == F13t(t^2 + t + 9)                                    6
True                                                            7
sage: A2 == F13t(6*t + 10)                                      8
True                                                            9
sage: A5 == F13t(7*t + 4)                                       10
True                                                            11
```

Combining this computation with the target polynomial we derived earlier, a quadratic arithmetic program associated to the rank-1 constraint system $R_{3.fac\_zk}$ is given by

$$QAP(R_{3.fac\_zk}) = \{x^2 + x + 9,$$
$$\{0, 0, 6x + 10, 0, 0, 7x + 4\}, \{0, 0, 0, 6x + 10, 7x + 4, 0\}, \{0, 7x + 4, 0, 0, 0, 6x + 10\}\}$$

QAP Satisfiability   One of the major points of quadratic arithmetic programs in proofing systems is that solutions of their associated rank-1 constraints systems are in 1:1 correspondence with certain polynomials P such that P is divisible by the target polynomial T of the QAP if and only if the solution id a solution. Verifying solutions to the R1CS and hence, checking proper circuit execution is then achievable by polynomial division of P by T. `clarify language`

To be more specific, let R be some rank-1 constraints system with associated assignment variables $(I_1, \ldots, I_n; W_1, \ldots, W_m)$ and let $QAP(R)$ be a quadratic arithmetic program of R. Then

the tuple $(I_1, \ldots, I_n; W_1, \ldots, W_m)$ is a solution to the R1CS if and only if the following polynomial is divisible by the target polynomial T:

$$P_{(I;W)} = \left(A_0 + \Sigma_j^n I_j \cdot A_j + \Sigma_j^m W_j \cdot A_{n+j}\right) \cdot \left(B_0 + \Sigma_j^n I_j \cdot B_j + \Sigma_j^m W_j \cdot B_{n+j}\right) - \left(C_0 + \Sigma_j^n I_j \cdot C_j + \Sigma_j^m W_j \cdot C_{n+j}\right) \qquad (6.13)$$

Every tuple $(I;W)$ defines a polynomial $P_{(I;W)}$, and, since each polynomial A_j, B_j and C_j is of degree $k-1$, $P_{(I;W)}$ is of degree $(k-1) \cdot (k-1) = k^2 - 2k + 1$.

To understand how quadratic arithmetic programs define formal languages, observe that every QAP over a field $\mathbb{F}$ defines a decision function over the alphabet $\Sigma_I \times \Sigma_W = \mathbb{F} \times \mathbb{F}$ in the following way:

$$R_{QAP} : \mathbb{F}^* \times \mathbb{F}^* \to \{true, false\} \; ; \; (I;W) \mapsto \begin{cases} true & P_{(I;W)} \text{ is divisible by } T \\ false & else \end{cases} \qquad (6.14)$$

Every QAP therefore defines a formal language, and, if the QAP is associated to an R1CS, it can be shown that the two languages are equivalent. A **statement** is a membership claim "There is a word $(I;W)$ in L_{QAP}". A proof to this claim is therefore a polynomial $P_{(I;W)}$, which is verified by dividing $P_{(I;W)}$ by T.

Note the structural similarity to the definition of an R1CS in XXX and the different ways of computing proofs in both systems. For circuits and their associated rank-1 constraints systems, a constructive proof consists of a valid assignment of field elements to the edges of the circuit, or the variables in the R1CS. However, in the case of QAPs, a valid proof consists of a polynomial $P_{(I;W)}$. `[add reference]`

To compute a proof for a statement in L_{QAP} given some instance I, a proofer first needs to compute a constructive proof W, e.g. by executing the circuit. With $(I;W)$ at hand, the proofer can then compute the polynomial $P_{(I;W)}$ and publish it as proof.

Verifying a constructive proof in the case of a circuit is achieved by executing the circuit, comparing the result to the given proof, and verifying the same proof in the R1CS picture means checking if the elements of the proof satisfy all equation.

In contrast, verifying a proof in the case of a QAP is done by polynomial division of the proof P by the target polynomial T of the QAP. The proof checks out if and only if P is divisible by T.

Example 123. Consider the quadratic arithmetic program $QAP(R_{3.fac\_zk})$ from example XXX and its associated R1CS from example XXX. To give an intuition of how proofs in the language $L_{QAP(R_{3.fac\_zk})}$ lets consider the instance $I_1 = 11$. As we know from example XXX, $(W_1, W_2, W_3, W_5) = (2, 3, 4, 6)$ is a proper witness, since $(I_1; W_1, W_2, W_3, W_5) = (11; 2, 3, 4, 6)$ is a valid circuit assignment and hence, a solution to $R_{3.fac\_zk}$ and a constructive proof for language $L_{R_{3.fac\_zk}}$. `[add reference]` `[add reference]` `[add reference]`

In order to transform this constructive proof into a membership proof in language $L_{QAP(R_{3.fac\_zk})}$ a proofer has to use the elements of the constructive proof, to compute the polynomial $P_{(I;W)}$.

In the case of $(I_1; W_1, W_2, W_3, W_5) = (11; 2, 3, 4, 6)$, the associated proof is computed as fol-

lows:

$$
\begin{aligned}
P_{(I;W)} &= \left(A_0+\textstyle\sum_j^n I_j\cdot A_j+\sum_j^m W_j\cdot A_{n+j}\right)\cdot\left(B_0+\sum_j^n I_j\cdot B_j+\sum_j^m W_j\cdot B_{n+j}\right)-\left(C_0+\sum_j^n I_j\cdot C_j+\sum_j^m W_j\cdot C_{n+j}\right) \\
&= (2(6x+10)+6(7x+4))\cdot(3(6x+10)+4(7x+4))-(11(7x+4)+6(6x+10)) \\
&= ((12x+7)+(3x+11))\cdot((5x+4)+(2x+3))-((12x+5)+(10x+8)) \\
&= (2x+5)\cdot(7x+7)-(9x) \\
&= (x^2+2\cdot 7x+5\cdot 7x+5\cdot 7)-(9x) \\
&= (x^2+x+9x+9)-(9x) \\
&= x^2+x+9
\end{aligned}
$$

Given instance $I_1 = 11$ a proofer therefore provides the polynomial x^2+x+9 as proof. To verify this proof, any verifier can then look up the target polynomial T from the QAP and divide $P_{(I;W)}$ by T. In this particular example, $P_{(I;W)}$ is equal to the target polynomial T, and hence, it is divisible by T with $P/T = 1$. The verification therefore checks the proof.

```
sage: F13 = GF(13)                                                    12
sage: F13t.<t> = F13[]                                                13
sage: T = F13t(t^2 + t + 9)                                           14
sage: P = F13t((2*(6*t+10)+6*(7*t+4))*(3*(6*t+10)+4*(7*t +4))         15
   -(11*(7*t+4)+6*(6*t+10)))
sage: P == T                                                         16
True                                                                17
sage: P % T # remainder                                             18
0                                                                   19
```

To give an example of a false proof, consider the tuple $(I_1;W_1,W_2,W_3,W_4) = (11,2,3,4,8)$. Executing the circuit, we can see that this is not a valid assignment and not a solution to the R1CS, and hence, not a constructive knowledge proof in $L_{3.fac\_zk}$. However, a proofer might use these values to construct a false proof $P_{(I;W)}$:

$$
\begin{aligned}
P'_{(I;W)} &= \left(A_0+\textstyle\sum_j^n I_j\cdot A_j+\sum_j^m W_j\cdot A_{n+j}\right)\cdot\left(B_0+\sum_j^n I_j\cdot B_j+\sum_j^m W_j\cdot B_{n+j}\right)-\left(C_0+\sum_j^n I_j\cdot C_j+\sum_j^m W_j\cdot C_{n+j}\right) \\
&= (2(6x+10)+8(7x+4))\cdot(3(6x+10)+4(7x+4))-(8(6x+10)+11(7x+4)) \\
&= 8x^2+6
\end{aligned}
$$

Given instance $I_1 = 11$, a proofer therefore provides the polynomial $8x^2+6$ as proof. To verify this proof, any verifier can look up the target polynomial T from the QAP and divide $P_{(I;W)}$ by T. However, polynomial division has a remainder

$$
(8x^2+6)/(x^2+x+9) = 8+\frac{5x+12}{x^2+x+9}
$$

This implies that $P_{(I;W)}$ is not divisible by T, and hence, the verification does not check the proof. Any verifier can therefore show that the proof is false.

```
sage: F13 = GF(13)                                                    20
sage: F13t.<t> = F13[]                                                21
sage: T = F13t(t^2 + t + 9)                                           22
sage: P = F13t((2*(6*t+10)+8*(7*t+4))*(3*(6*t+10)+4*(7*t+4))-(        23
   8*(6*t+10)+11*(7*t+4)))
```

```
sage: P == F13t(8*t^2 + 6)                                          24
True                                                                25
sage: P % T # remainder                                             26
5*t + 12                                                            27
```

Bibliography

Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, 2016:260, 2016. URL `http://eprint.iacr.org/2016/260`.

David Fifield. The equivalence of the computational diffie–hellman and discrete logarithm problems in certain groups, 2012. URL `https://web.stanford.edu/class/cs259c/finalpapers/dlp-cdh.pdf`.

Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46766-3. URL `https://fmouhart.epheme.re/Crypto-1617/TD08.pdf`.

Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. Cryptology ePrint Archive, Report 2016/492, 2016. `https://ia.cr/2016/492`.