
1 Operational notes

2 Document updated on **March 19, 2022**.

















































3 The following colors are **not** part of the final product, but serve as highlights in the edit-
4 ing/review process:

- 5 • text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan
- 6 • terms that have not yet been defined in the book
- 7 • text that needs advice from the communications/marketing team: Aaron & Shane
- 8 • text that needs to be completed or otherwise edited (by Sylvia)

















































9 NB: This PDF only includes the Circuit Compilers chapter









































Todo list

11	zero-knowledge proofs	12
12	played with	12
13	finite field	12
14	elliptic curve	12
15	Update reference when content is finalized	12
16	methatical	12
17	numerical	12
18	a list of additional exercises	13
19	think about them	13
20	add some more informal explanation of absolute value	14
21	We haven't really talked about what a ring is at this point	14
22	What's the significance of this distinction?	15
23	reverse	15
24	Turing machine	15
25	polynomial time	15
26	sub-exponentially, with $\mathcal{O}((1 + \varepsilon)^n)$ and some $\varepsilon > 0$	15
27	Add text	16
28	\mathbb{Q} of fractions	16
29	Division in the usual sense is not defined for integers	16
30	Add more explanation of how this works	17
31	pseudocode	18
32	modular arithmetics	18
33	actual division	18
34	multiplicative inverses	18
35	factional numbers	18
36	exponentiation function	20
37	See XXX	20
38	once they accept that this is a new kind of calculations, its actually not that hard	20
39	perform Euclidean division on them	20
40	This Sage snippet should be described in more detail.	21
41	prime fields	23
42	residue class rings	23
43	Algorithm sometimes floated to the next page, check this for final version	23
44	Add a number and title to the tables	25
45	(-1) should be $(-a)$?	26
46	we have	28
47	rephrase	32
48	subtrahend	33
49	minuend	33

50	 what does this mean?	37
51	 Chapter 1?	126
52	 "rigorous"?	126
53	 "proving"?	126
54	 Add example	127
55	 Add more explanation	127
56	 I'd delete this, too distracting	127
57	 binary tuples	127
58	 add reference	128
59	 add reference	128
60	 check reference	128
61	 check reference	128
62	 Are we using w and x interchangeably or is there a difference between them?	129
63	 check reference	129
64	 jubjub	129
65	 Edwards form	129
66	 add reference	129
67	 add reference	129
68	 check wording	129
69	 add reference	129
70	 check references	130
71	 add reference	130
72	 add reference	130
73	 preimage	131
74	 check reference	131
75	 add reference	131
76	 check reference	132
77	 check reference	132
78	 add reference	133
79	 Can we reword this? It's grammatically correct but hard to read	133
80	 add reference	134
81	 Schur/Hadamard product	134
82	 add reference	134
83	 check reference	134
84	 check reference	135
85	 add reference	136
86	 check reference	137
87	 check reference	137
88	 check reference	137
89	 check reference	137
90	 check reference	138
91	 add reference	138
92	 add reference	139
93	 check reference	139
94	 check reference	139
95	 add reference	140
96	 add reference	140
97	 add reference	141

98	■ We already said this in this chapter	143
99	■ check reference	143
100	■ add reference	143
101	■ check reference	144
102	■ add reference	144
103	■ check reference	144
104	■ Should we refer to R1CS satisfiability (p. 137 here?	145
105	■ add reference	146
106	■ add reference	146
107	■ add reference	146
108	■ add reference	147
109	■ check reference	147
110	■ check reference	148
111	■ check reference	150
112	■ add reference	151
113	■ "by"?	151
114	■ add reference	151
115	■ check reference	151
116	■ add reference	151
117	■ add reference	151
118	■ check reference	151
119	■ add reference	151
120	■ clarify language	153
121	■ add reference	154
122	■ add reference	154
123	■ add reference	154
124	■ add reference	154
125	■ add references	157
126	■ add references to these languages?	157
127	■ add reference	160
128	■ add reference	161
129	■ add reference	161
130	■ add reference	162
131	■ add reference	163
132	■ add reference	163
133	■ add reference	165
134	■ add reference	165
135	■ add reference	166
136	■ add reference	166
137	■ add reference	166
138	■ add reference	166
139	■ add reference	166
140	■ add reference	167
141	■ add reference	167
142	■ add reference	167
143	■ add reference	167
144	■ add reference	167
145	■ add reference	168

146	 add reference	169
147	 "constraints" or "constrained"?	169
148	 add reference	170
149	 "constraints" or "constrained"?	170
150	 add reference	170
151	 "constraints" or "constrained"?	170
152	 add reference	171
153	 add reference	171
154	 add reference	171
155	 add reference	171
156	 add reference	172
157	 add reference	173
158	 add reference	173
159	 add reference	173
160	 shift	175
161	 bishift	176
162	 add reference	177
163	 add reference	178
164	 something missing here?	179
165	 suggar	180
166	 add reference	180
167	 add reference	181
168	 add reference	182
169	 add reference	182
170	 add reference	182
171	 add reference	183
172	 add reference	183
173	 add reference	183
174	 add reference	184
175	 add reference	185
176	 add reference	186
177	 add reference	186
178	 add reference	186
179	 add reference	187
180	 add reference	187
181	 add reference	187
182	 add reference	187
183	 add reference	187
184	 "invariable"?	187
185	 add reference	188
186	 add reference	188
187	 add reference	188
188	 add reference	189
189	 add reference	189
190	 add reference	190
191	 add reference	191
192	 add reference	191
193	 add reference	192

194	 add reference	192
195	 add reference	192
196	 add reference	192
197	 add reference	192
198	 add reference	193
199	 add reference	193
200	 add reference	193
201	 add reference	193
202	 add reference	193
203	 add reference	193
204	 add reference	193
205	 add reference	193
206	 add reference	193
207	 add reference	194
208	 add reference	194
209	 add reference	194
210	 add reference	194
211	 add reference	196
212	 add reference	196
213	 add reference	196
214	 add reference	196
215	 add reference	196
216	 add reference	196
217	 add reference	197
218	 add reference	197
219	 add reference	197
220	 add reference	197
221	 add reference	197
222	 add reference	198
223	 add reference	198
224	 add reference	198
225	 add reference	198
226	 add reference	199
227	 add reference	199
228	 add reference	199
229	 add reference	199
230	 add reference	199
231	 add reference	199
232	 add reference	199
233	 add reference	199

234

MoonMath manual

235

TechnoBob and the Least Scruples crew

236

March 19, 2022

Contents

238	1 Introduction	5
239	1.1 Target audience	5
240	1.2 The Zoo of Zero-Knowledge Proofs	6
241	To Do List	8
242	Points to cover while writing	8
243	2 Preliminaries	9
244	2.1 Preface and Acknowledgements	9
245	2.2 Purpose of the book	9
246	2.3 How to read this book	10
247	2.4 Cryptological Systems	10
248	2.5 SNARKS	10
249	2.6 complexity theory	10
250	2.6.1 Runtime complexity	10
251	2.7 Software Used in This Book	11
252	2.7.1 Sagemath	11
253	3 Arithmetics	12
254	3.1 Introduction	12
255	3.1.1 Aims and target audience	12
256	3.1.2 The structure of this chapter	13
257	3.2 Integer Arithmetics	13
258	Euclidean Division	16
259	The Extended Euclidean Algorithm	18
260	3.3 Modular arithmetic	19
261	Congurency	20
262	Modular Arithmetics	20
263	The Chinese Remainder Theorem	23
264	Modular Inverses	26
265	3.4 Polynomial Arithmetics	29
266	Polynomial Arithmetics	33
267	Euklidean Division	34
268	Prime Factors	36
269	Lange interpolation	37
270	4 Algebra	40
271	4.1 Groups	40
272	Commutative Groups	41
273	Finite groups	43

274		Generators	43
275		The discrete Logarithm problem	43
276	4.1.1	Cryptographic Groups	44
277		The discret logarithm assumption	45
278		The decisional Diffi Hellman assumption	47
279		The computational Diffi Hellman assumption	47
280		Cofactor Clearing	48
281	4.1.2	Hashing to Groups	48
282		Hash functions	48
283		Hashing to cyclic groups	50
284		Hashing to modular arithmetics	51
285		Pederson Hashes	55
286		MimC Hashes	55
287		Pseudo Random Functions in DDH-A groups	55
288	4.2	Commutative Rings	55
289		Hashing to Commutative Rings	58
290	4.3	Fields	58
291		Prime fields	60
292		Square Roots	61
293		Exponentiation	63
294		Hashing into Prime fields	63
295		Extension Fields	63
296		Hashing into extension fields	66
297	4.4	Projective Planes	67
298	5	Elliptic Curves	69
299	5.1	Elliptic Curve Arithmetics	69
300	5.1.1	Short Weierstraß Curves	69
301		Affine short Weierstraß form	70
302		Affine compressed representation	74
303		Affine group law	75
304		Scalar multiplication	80
305		Projective short Weierstraß form	83
306		Projective Group law	85
307		Coordinate Transformations	85
308	5.1.2	Montgomery Curves	85
309		Affine Montgomery Form	87
310		Affine Montgomery coordinate transformation	88
311		Montgomery group law	90
312	5.1.3	Twisted Edwards Curves	90
313		Twisted Edwards Form	91
314		Twisted Edwards group law	92
315	5.2	Elliptic Curves Pairings	93
316		Embedding Degrees	93
317		Elliptic Curves over extension fields	95
318		Full Torsion groups	96
319		Torsion-Subgroups	98
320		The Weil Pairing	100

321	5.3	Hashing to Curves	103
322		Try and increment hash functions	103
323	5.4	Constructing elliptic curves	106
324		The Trace of Frobenius	106
325		The j -invariant	107
326		The Complex Multiplication Method	108
327		The <i>BLS6_6</i> pen& paper curve	117
328		Hashing to the pairing groups	124
329	6	Statements	126
330	6.1	Formal Languages	126
331		Decision Functions	127
332		Instance and Witness	130
333		Modularity	133
334	6.2	Statement Representations	133
335	6.2.1	Rank-1 Quadratic Constraint Systems	133
336		R1CS representation	134
337		R1CS Satisfiability	136
338		Modularity	138
339	6.2.2	Algebraic Circuits	138
340		Algebraic circuit representation	138
341		Circuit Execution	143
342		Circuit Satisfiability	145
343		Associated Constraint Systems	146
344	6.2.3	Quadratic Arithmetic Programs	151
345		QAP representation	151
346		QAP Satisfiability	153
347	7	Circuit Compilers	157
348	7.1	A Pen-and-Paper Language	157
349	7.1.1	The Grammar	157
350	7.1.2	The Execution Phases	159
351		The Setup Phase	159
352		The Prover Phase	161
353	7.2	Common Programing concepts	161
354	7.2.1	Primitive Types	161
355		The base-field type	161
356		The Subtraction Constraint System	165
357		The Inversion Constraint System	166
358		The Division Constraint System	167
359		The boolean Type	168
360		The boolean Constraint System	168
361		The AND operator constraint system	169
362		The OR operator constraint system	169
363		The NOT operator constraint system	170
364		Modularity	171
365		Arrays	174
366		The Unsigned Integer Type	174

367		The uN Constraint System	175
368		The Unsigned Integer Operators	176
369	7.2.2	Control Flow	177
370		The Conditional Assignment	177
371		Loops	179
372	7.2.3	Binary Field Representations	180
373	7.2.4	Cryptographic Primitives	182
374		Twisted Edwards curves	182
375		Twisted Edwards curves constraints	182
376		Twisted Edwards curve addition	183
377	8	Zero Knowledge Protocols	184
378	8.1	Proof Systems	184
379	8.2	The “Groth16” Protocol	185
380		The Setup Phase	187
381		The Proofer Phase	192
382		The Verification Phase	195
383		Proof Simulation	197
384	9	Exercises and Solutions	200

Chapter 7

Circuit Compilers

As we have seen in the previous chapter, statements can be formalized as membership or knowledge claims in formal language, and algebraic circuits as well as rank-1 constraint systems are two practically important ways to define those languages.

However, both algebraic circuits and rank-1 constraint systems are not ideal from a developers point of view, because they deviate substantially from common programming paradigms. Writing real-world applications as circuits and the associated verification in terms of rank-1 constraint systems is at least as troublesome as writing any other low-level language like assembler code. To allow for complex statement design, it is therefore necessary to have some kind of compiler framework, capable of transforming high-level languages into arithmetic circuits and associated rank-1 constraint systems.

As we have seen in XXX as well as XXX and XXX, both arithmetic circuits and rank-1 constraint systems have a modularity property by which it is possible to synthesize complex circuits from simple ones. A basic approach taken by many circuit/R1CS compilers is therefore to provide a library of atomic and simple circuits and then define a way to combine those basic building blocks into arbitrary complex systems.

In this chapter, we provide an introduction to basic concepts of so-called **circuit compilers** and derive a toy language which we can “compile” in a pen-and-paper approach into algebraic circuits and their associated rank-1 constraint systems.

We start with a general introduction to our language, and then introduce atomic types like booleans and unsigned integers. Then we define the fundamental control flow primitives like the if-then-else conditional and the bounded loop. We will look at basic functionality primitives like elliptic curve cryptography. Primitives like these are often called **gadgets** in the literature.

7.1 A Pen-and-Paper Language

To explain basic concepts of circuit compilers and their associated high-level languages, we derive an informal toy language and associated “brain-compiler” which we name PAPER (**Pen-And-Paper Execution Rules**). PAPER allows programmers to define statements in Rust-like pseudo-code. The language is inspired by ZOKRATES and `circom`.

7.1.1 The Grammar

In PAPER, any statement is defined as an ordered list of functions, where any function has to be declared in the list before it is called in another function of that list. The last entry in a statement has to be a special function, called `main`. Functions take a list of typed parameters as inputs

add references

add references to these languages?

and compute a tuple of typed variables as output, where types are special functions that define how to transform that type into another type, ultimately transforming any type into elements of the base field where the circuit is defined over.

Any statement is parameterized over the field that the circuit will be defined on, and has additional optional parameters of unsigned type, needed to define the size of array or the counter of bounded loops. The following definition makes the grammar of a statement precise using a command line language like description:

```
statement <Name> {F:<Field> [ , <N_1: unsigned>, ... ] } {
  [fn <Name>([[pub]<Arg>:<Type>, ...]) -> (<Type>, ...)] {
    [let [pub] <Var>:<Type> ; ... ]
    [let const <Const>:<Type>=<Value> ; ... ]
    Var<==>(fn ([<Arg>|<Const>|<Var>, ...]) | (<Arg>|<Const>|<Var>)) ;
    return (<Var>, ...) ;
  } ; ...]
  fn main([[pub]<Arg>:<Type>, ...]) -> (<Type>, ...) {
    [let [pub] <Var>:<Type> ; ... ]
    [let const <Const>:<Type>=<Value> ; ... ]
    Var<==>(fn ([<Arg>|<Const>|<Var>, ...]) | (<Arg>|<Const>|<Var>)) ;
    return (<Var>, ...) ;
  } ;
}
```

Function arguments and variables are private by default, but can be declared as public by the `pub` specifier. Declaring arguments and variables as public always overwrites any previous or conflicting private declarations. Every argument, constant or variable has a type, and every type is defined as a function that transforms that type into another type:

```
type <TYPE>( t1 : <TYPE_1>) -> TYPE_2{
  let t2: TYPE_2 <== fn(TYPE_1)
  return t2
}
```

Many real-world circuit languages are based on a similar, but of course more sophisticated approach than PAPER. The purpose of PAPER is to show basic principles of circuit compilers and their associated high-level languages.

Example 124. To get a better understanding of the grammar of PAPER, the following constitutes proper high-level code that follows the grammar of the PAPER language, assuming that all types in that code have been defined elsewhere.

```
statement MOCK_CODE {F: F_43, N_1 = 1024, N_2 = 8} {
  fn foo(in_1 : F, pub in_2 : TYPE_2) -> F {
    let const c_1 : F = 0 ;
    let const c_2 : TYPE_2 = SOME_VALUE ;
    let pub out_1 : F ;
    out_1<== c_1 ;
    return out_1 ;
  } ;

  fn bar(pub in_1 : F) -> F {
    let out_1 : F ;
    out_1<==foo(in_1);
    return out_1 ;
  } ;
}
```

```

4397     } ;
4398
4399     fn main(in_1 : TYPE_1) -> (F, TYPE_2) {
4400         let const c_1 : TYPE_1 = SOME_VALUE ;
4401         let const c_2 : F = 2 ;
4402         let const c_3 : TYPE_2 = SOME_VALUE ;
4403         let pub out_1 : F ;
4404         let out_2 : TYPE_2 ;
4405         c_1 <== in_1 ;
4406         out_1 <== foo(c_2) ;
4407         out_2 <== TYPE_2 ;
4408         return (out_1, out_2) ;
4409     } ;
4410 }

```

7.1.2 The Execution Phases

In contrast to normal executable programs, programs for circuit compilers have two modes of execution. The first mode, usually called the **setup phase**, is executed in order to generate the circuit and its associated rank-1 constraint system, the latter of which is then usually used as input to some zero-knowledge proof system.

The second mode of execution is usually called the **prover phase**. In this phase, a prover usually computes a valid assignment to the circuit. Depending on the use case, this valid assignment is then either directly used as constructive proof for proper circuit execution or is transferred as input to the proof generation algorithm of some zero-knowledge proof system, where the full-sized, non hiding constructive proof is processed into a succinct proof with various levels of zero knowledge.

Modern circuit languages and their associated compilers abstract over those two phases and provide a unified **interphase** to the developer, who then writes a single program that can be used in both phases.

To give the reader a clear, conceptual distinction between the two phases, PAPER keeps them separated. Code can be “brain-compiled” during the **setup-phase** in a pen-and-paper approach into visual circuits. Once a circuit is derived, it can be executed in a **prover phase** to generate a valid assignment. The valid assignment is then interpreted as a constructive proof for a knowledge claim in the associated language.

The Setup Phase In PAPER, the task of the setup phase is to compile code in the PAPER language into a visual representation of an algebraic circuit. Deriving the circuit from the code in a pen-and-paper style is what we call **brain compiling**.

Given some statement description that adheres to the correct grammar, we start circuit development with an empty circuit, compile the main function first and then inductively compile all other functions as they are called during the process.

For every function we compile, we draw a box-node for every argument, every variable and every constant of that function. If the node represents a variable, we label it with that variable’s name, and if it represents a constant, we label it with that constant’s value. We group arguments into a subgraph labeled “inputs” and return values into a subgraph labeled “outputs”. We then group everything into a subgraph and label that subgraph with the function’s name.

After this is done, we have to do a consistency and type check for every occurrence of the

assignment operator `<==`. We have to ensure that the expression on the right side of the operator is well defined and that the types of both side match.

Then we compile the right side of every occurrence of the assignment operator `<==`. If the right side is a constant or variable defined in this function, we draw a dotted line from the box-node that represents the left side of `<==` to the box node that represents the right side of the same operator. If the right side represents an argument of that function we draw a line from the box-node that represents the left side of `<==` to the box node that represents the right side of the same operator.

If the right side of the `<==` operator is a function, we look into our database, find its associated circuit and draw it. If no circuit is associated to that function yet, we repeat the compilation process for that function, drawing edges from the function's argument to its input nodes and from the functions output nodes to the nodes on the right side of `<==`.

During that process, edge labels are drawn according to the rules from XXX. If the associated variable represents a private value, we use the *W* label to indicate a witness, and if it represents a public value, we use the *I* label to indicate an instance.

add reference

Once this is done, we compile all occurring types in a function, by compiling the function of each type. We do this inductively until we reach the type of the base field. Circuits have no notion of types, only of field elements; hence, every type needs to be compiled to the field type in a sequence of compilation steps.

The compilation stops once we have inductively replaced all functions by their circuits. The result is a circuit that contains many unnecessary box nodes. In a final optimization step, all box nodes that are directly linked to each other are collapsed into a single node, and all box nodes that represent the same constants are collapsed into a single node.

Of course, PAPER's brain-compiler is not properly defined in any formal manner. Its purpose is to highlight important steps that real-world compilers undergo in their setup phases.

Example 125 (A trivial Circuit). To give an intuition of how to write and compile circuits in the PAPER language, consider the following statement description:

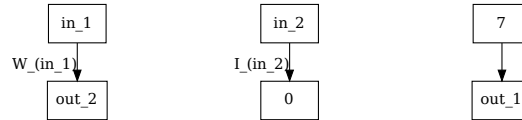
```
statement trivial_circuit {F:F_13} {
  fn main{F}(in1 : F, pub in2 : F) -> (F,F) {
    let const outc1 : F = 0 ;
    let const incl : F = 7 ;
    let out1 : F ;
    let out2 : F ;
    out1 <== incl;
    out2 <== in1;
    outc1 <== in2;
    return (out1, out2) ;
  }
}
```

To brain-compile this statement into an algebraic circuit with PAPER, we start with an empty circuit and evaluate function `main`, which is the only function in this statement.

We draw box-nodes for every argument, every constant and every variable of the function and label them with their names or values, respectively. Then we do a consistency and type check for every `<==` operator in the function. Since the circuit only wires inputs to outputs and all elements have the same type, the check is valid.

Then we evaluate the right side of the assignment operators. Since, in our case, the right side of each operator is not a function, we draw edges from the box-nodes on the right side to the associated box node on the left side. To label those edges, we use the general rules of algebraic

circuits as defined in XXX. According to those rules, every incoming edge of a sink node has a label and every outgoing edge of a source node has a label, if the node is labeled with a variable. Since nodes that represent constants are implicitly assumed to be private, and since the public specifier determines if an edge is labeled with W or I , we get the following circuit:



The Prover Phase In PAPER, a so-called **prover phase** can be executed once the setup phase has generated a circuit image from its associated high-level code. This is done by executing the circuit while assigning proper values to all input nodes of the circuit. However, in contrast to most real-world compilers, PAPER does not tell the prover how to find proper input values to a given circuit. Real-world programming languages usually provide this data by computations that are done outside of the circuit.

Example 126. Consider the circuit from example XXX. Valid assignments to this circuit are constructive proofs that the pair of inputs (S_1, S_2) is a point on the tiny-jubjub curve. However, the circuit does not provide a way to actually compute proper values for S_1 and S_2 . Any real-world system therefore needs an auxiliary computation that provides those values.

7.2 Common Programing concepts

In this section, we cover concepts that appear in almost every programming language, and see how they can be implemented in circuit compilers.

7.2.1 Primitive Types

Primitive data types like booleans, (unsigned) integers, or strings are the most basic building blocks one can expect to find in every general high-level programming language. In order to write statements as computer programs that compile into circuits, it is therefore necessary to implement primitive types as constraint systems, and define their associated operations as circuits.

In this section, we look at some common ways to achieve this. After a recapitulation of the atomic type of prime field elements, we start with an implementation of the boolean type and its associated boolean algebra as circuits. After that, we define unsigned integers based on the boolean type, and leave the implementation of signed integers as an exercise to the reader.

It should be noted, however, that while primitive data types in common programming languages (like C, Go, or Rust) have a one-to-one correspondence with objects in the computer's memory, this is not the case for most languages that compile into algebraic circuits. As we will see in the following paragraphs, common primitives like booleans or unsigned integers require many constraints and memory. Primitives different from the underlying field elements can be expensive.

The base-field type

Since both algebraic circuits and their associated rank-1 constraint systems are defined over a finite field, elements from that field are the atomic informational units in those models. In this

sense, field elements $x \in \mathbb{F}$ are for algebraic circuits what bits are for computers.

In PAPER, we write F for this type and specify the actual field instance for every statement in curly brackets after the name of that statement. Two functions are associated to this type, which are induced by the **addition** and **multiplication** law in the field F . We write

$$\text{MUL} : F \times F \rightarrow F ; (x, y) \mapsto \text{MUL}(x, y) \quad (7.1)$$

$$\text{ADD} : F \times F \rightarrow F ; (x, y) \mapsto \text{ADD}(x, y) \quad (7.2)$$

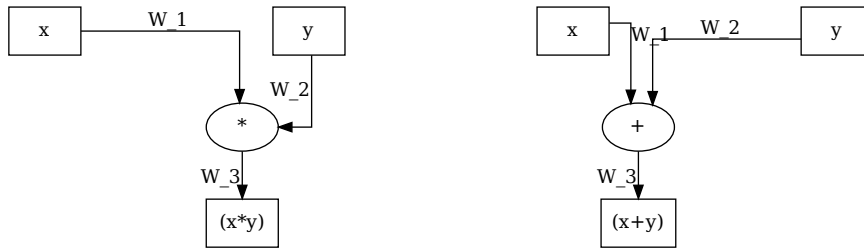
Circuit compilers have to compile these functions into algebraic gates, as explained in XXX. Every other function has to be expressed in terms of them and proper wiring.

To represent addition and multiplication in the PAPER language, we define the following two functions:

```
fn MUL(x : F, y : F) -> (MUL(x, y) : F) { }
```

```
fn ADD(x : F, y : F) -> (ADD(x, y) : F) { }
```

The compiler then compiles every occurrence of the MUL or the ADD function into the following circuits:



Example 127 (Basic gates). To give an intuition of how a real-world compiler might transform addition and multiplication in algebraic expressions into a circuit, consider the following PAPER statement:

```
statement basic_ops {F:F_13} {
  fn main(in_1 : F, pub in_2 : F) -> (out_1:F, out_2:F) {
    out_1 <== MUL(in_1, in_2) ;
    out_2 <== ADD(in_1, in_2) ;
  }
}
```

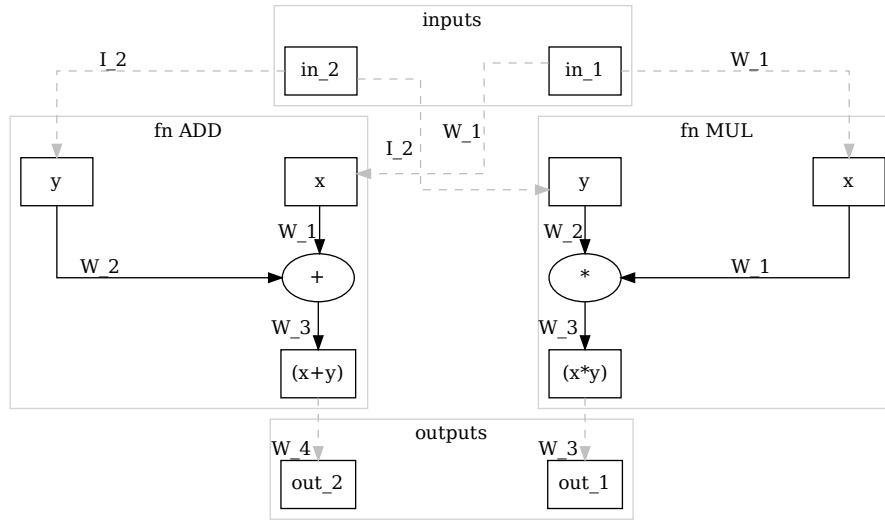
To compile this into an algebraic circuit, we start with an empty circuit and evaluate the function `main`, which is the only function in this statement.

We draw an inputs subgraph containing box-nodes for every argument of the function, and an outputs subgraph containing box-nodes for every factor in the return value. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

We check the validity of every expression on the right side of every `<==` operator including a type check. In our case, every variable is of the `field` type and hence the types match the types of the `MUL` as well as the `ADD` function and the type of the left sides of `<==` operators.

We evaluate the expressions on the right side of every `<==` operator inductively, replacing every occurrence of a function with a subgraph that represents its associated circuit.

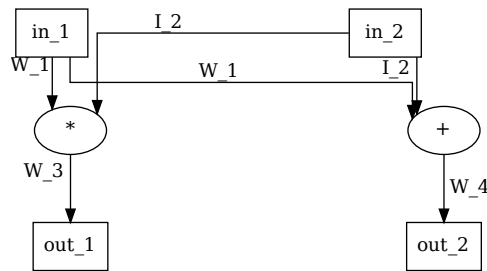
4560 According to PAPER, every occurrence of the `public` specifier overwrites the associate
 4561 `private` default value. Using the appropriate edge labels we get:



4562

4563 Any real-world compiler might process its associated high-level language in a similar way,
 4564 replacing functions, or gadgets by predefined associated circuits. This process is often followed
 4565 by various optimization steps that try to reduce the number of constraints as much as possible.

4566 In PAPER, we optimize this circuit by collapsing all box nodes that are directly connected
 4567 to other box nodes, adhering to the rule that a variable's `public` specifier overwrites any
 4568 `private` specifier. Reindexing edge labels, we get the following circuit as our pen and pencil
 4569 compiler output:



4570

4571 *Example 128 (3-factorization).* Consider our 3-factorization problem from example XXX and
 4572 the associated circuit $C_{3.fac_zk}(\mathbb{F}_{13})$ we provided in example XXX. To understand the process of
 4573 replacing high-level functions by their associated circuits inductively, we want define a PAPER
 4574 statement that we brain-compile into an algebraic circuit equivalent to $C_{3.fac_zk}(\mathbb{F}_{13})$. We write

```
4575 statement 3_fac_zk {F:F_13} {
4576   fn main(x_1 : F, x_2 : F, x_3 : F) -> (pub 3_fac_zk : F) {
4577     f_3.fac_zk <== MUL( MUL( x_1 , x_2 ) , x_3 ) ;
4578   }
4579 }
```

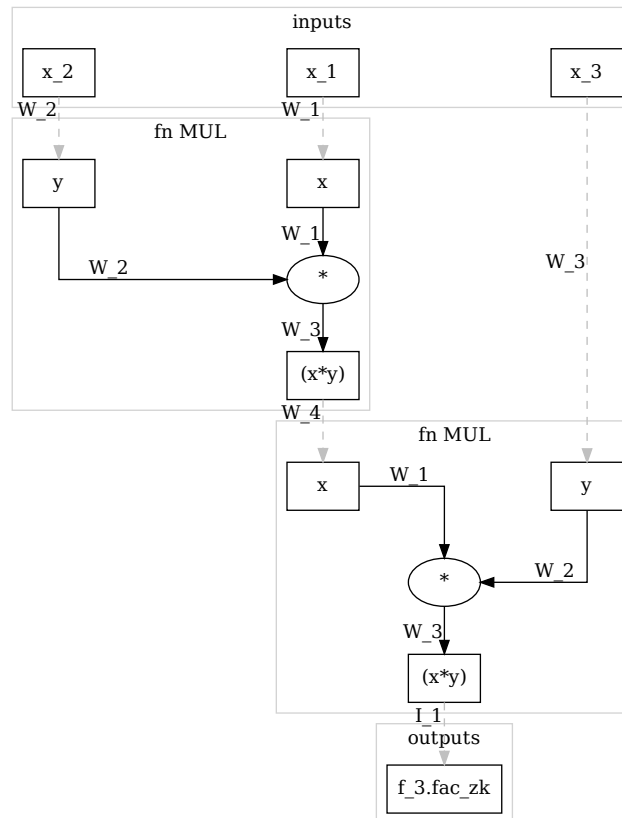
add refer-
ence

add refer-
ence

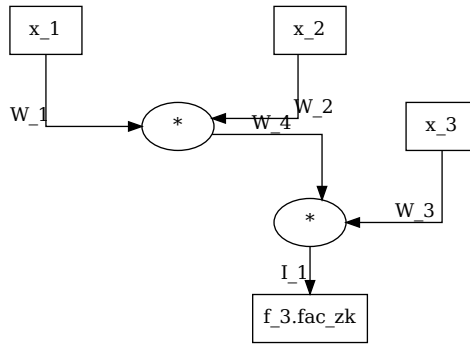
Using PAPER, we start with an empty circuit and then add 3 input nodes to the input subgraph as well as 1 output node to the output subgraph. All these nodes are decorated with the associated variable names. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

We check the validity of every expression on the right side of the single `<==` operator including a type check.

We evaluate the expressions on the right side of every `<==` operator inductively. We have two nested multiplication functions and we replace them by the associated multiplication circuits, starting with the most outer function. We get:



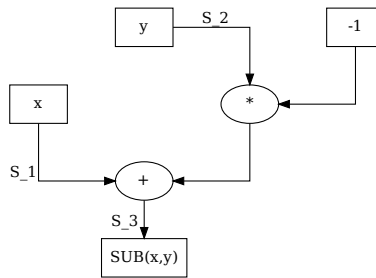
In a final optimization step, we collapse all box nodes directly connected to other box nodes, adhering to the rule that a variable's `public` specifier overwrites any `private` specifier. Reindexing edge labels we get the following circuit:



4593

4594 **The Subtraction Constraint System** By definition, algebraic circuits only contain addition
 4595 and multiplication gates, and it follows that there is no single gate for field subtraction, despite
 4596 the fact that subtraction is a native operation in every field.

4597 High-level languages and their associated circuit compilers, therefore, need another way to
 4598 deal with subtraction. To see how this can be achieved, recall that subtraction is defined by addi-
 4599 tion with the additive inverse, and that the inverse can be computed efficiently by multiplication
 4600 with -1 . A circuit for field subtraction is therefore given by



4601

4602 Using the general method from XXX, the circuits associated rank-1 constraint system is given
 4603 by:

$$(S_1 + (-1) \cdot S_2) \cdot 1 = S_3 \quad (7.3)$$

4604 Any valid assignment $\{S_1, S_2, S_3\}$ to this circuit therefore enforces the value S_3 to be the differ-
 4605 ence $S_1 - S_2$.

4606 Real-world compilers usually provide a gadget or a function to abstract over this circuit
 4607 such that programers can use subtraction as if it were native to circuits. In PAPER, we define
 4608 the following subtraction function that compiles to the previous circuit:

```
4609 fn SUB(x : F, y : F) -> (SUB(x,y) : F) {
4610   constant c : F = -1 ;
4611   SUB <== ADD(x , MUL( y , c ) );
4612 }
```

4613 In the setup phase of a statement, we compile every occurrence of the SUB function into an
 4614 instance of its associated subtraction circuit, and edge labels are generated according to the
 4615 rules from XXX.

add refer-
enceadd refer-
ence

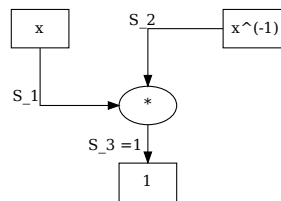
The Inversion Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field inversion, despite the fact that inversion is a native operation in every field.

If the underlying field is a prime field, one approach would be to use Fermat’s little theorem [XXX](#) to compute the multiplicative inverse inside the circuit. To see how this works, let \mathbb{F}_p be the prime field. The multiplicative inverse x^{-1} of a field element $x \in \mathbb{F}$ with $x \neq 0$ is then given by $x^{-1} = x^{p-2}$, and computing x^{p-2} in the circuit therefore computes the multiplicative inverse.

Unfortunately, real-world primes p are large and computing x^{p-2} by repeated multiplication of x with itself is infeasible. A “double and multiply” approach (as described in [XXX](#)) is faster, as it computes the power in roughly $\log_2(p)$ steps, but still adds a lot of constraints to the circuit.

Computing inverses in the circuit makes no use of the fact that inversion is a native operation in any field. A more constraints friendly approach is therefore to compute the multiplicative inverse outside of the circuit and then only enforce correctness of the computation in the circuit.

To understand how this can be achieved, observe that a field element $y \in \mathbb{F}$ is the multiplicative inverse of a field element $x \in \mathbb{F}$ if and only if $x \cdot y = 1$ in \mathbb{F} . We can use this, and define a circuit that has two inputs, x and y , and enforces $x \cdot y = 1$. It is then guaranteed that y is the multiplicative inverse of x . The price we pay is that we can not compute y by circuit execution, but auxiliary data is needed to tell any prover which value of y is needed for a valid circuit assignment. The following circuit defines the constraint



Using the general method from [XXX](#), the circuit is transformed into the following rank-1 constraint system:

$$S_1 \cdot S_2 = 1 \quad (7.4)$$

Any valid assignment $\{S_1, S_2\}$ to this circuit enforces that S_2 is the multiplicative inverse of S_1 , and, since there is no field element S_2 such that $0 \cdot S_2 = 1$, it also handles the fact that the multiplicative inverse of 0 is not defined in any field.

Real-world compilers usually provide a gadget or a function to abstract over this circuit, and those functions compute the inverse x^{-1} as part of their witness generation process. Programmers then don’t have to care about providing the inverse as auxiliary data to the circuit. In [PAPER](#), we define the following inversion function that compiles to the previous circuit:

```

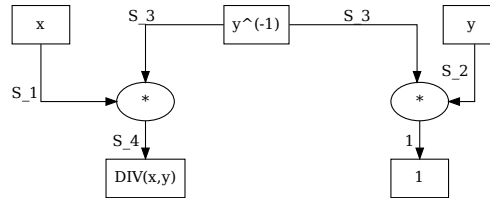
4645 fn INV(x : F, y : F) -> (x_inv : F) {
4646   constant c : F = 1 ;
4647   c <== MUL( x , y ) ;
4648   x_inv <== y ;
4649 }
```

As we see, this functions takes two inputs, the field value and its inverse. It therefore does not handle the computation of the inverse by itself. This is to keep [PAPER](#) as simple as possible.

In the setup phase, we compile every occurrence of the INV function into an instance of the inversion circuit [XXX](#), and edge labels are generated according to the rules from [XXX](#).

The Division Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field division, despite the fact that division is a native operation in every field.

Implementing division as a circuit, we use the fact that division is multiplication with the multiplicative inverse. We therefore define division as a circuit using the inversion circuit and constraint system from the previous paragraph. Expensive inversion is computed outside of the circuit and then provided as circuit input. We get



Using the method from XXX, we transform this circuit into the following rank-1 constraint system:

$$\begin{aligned} S_2 \cdot S_3 &= 1 \\ S_1 \cdot S_3 &= S_4 \end{aligned}$$

Any valid assignment $\{S_1, S_2, S_3, S_4\}$ to this circuit enforces S_4 to be the field division of S_1 by S_2 . It handles the fact that division by 0 is not defined, since there is no valid assignment in case $S_2 = 0$.

In PAPER, we define the following division function that compiles to the previous circuit:

```
fn DIV(x : F, y : F, y_inv : F) -> (DIV : F) {
  DIV <== MUL( x , INV( y, y_inv ) ) ;
}
```

In the setup phase, we compile every occurrence of the binary INV operator into an instance of the inversion circuit.

Exercise 44. Let F be the field \mathbb{F}_5 of modular 5 arithmetics from example XXX. Brain compile the following PAPER statement into an algebraic circuit:

```
statement STUPID_CIRC {F: F_5} {
  fn foo(in_1 : F, in_2 : F)->(out_1 : F, out_2 : F){
    constant c_1 : F = 3 ;
    out_1<== ADD( MUL( c_1 , in_1 ) , in_1 ) ;
    out_2<== INV( c_1 , in_2 ) ;
  } ;

  fn main(in_1 : F, in_2 : F)->(out_1 : F, out_2 : TYPE_2){
    constant (c_1,c_2) : (F,F) = (3,2) ;
    (out_1,out_2) <== foo(in_1, in_2) ;
  } ;
}
```

Exercise 45. Consider the tiny-jubjub curve from example XXX and its associated circuit XXX. Write a statement in PAPER that brain-compiles the statement into a circuit equivalent to the one derived in XXX, assuming that curve points are instances and every other assignment is a witness.

4689 *Exercise 46.* Let $F = \mathbb{F}_{13}$ be the modular 13 prime field and $x \in F$ some field element. Define a
 4690 statement in PAPER such that given instance x a field element $y \in F$ is a witness for the statement
 4691 if and only if y is the square root of x .

4692 Brain compile the statement into a circuit and derive its associated rank-1 constraint system.
 4693 Consider the instance $x = 9$ and compute a constructive proof for the statement.

4694 The boolean Type

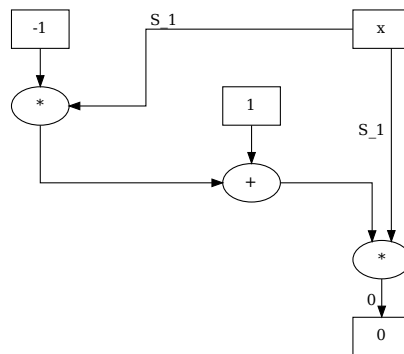
4695 booleans are a classical primitive type, implemented by virtually every higher programing lan-
 4696 guage. It is therefore important to implement booleans in circuits. One of the most common
 4697 ways to do this is by interpreting the additive and multiplicative neutral element $\{0, 1\} \subset \mathbb{F}$ as
 4698 the two boolean values such that 0 represents *false* and 1 represents *true*. boolean operators
 4699 like *and*, *or*, or *xor* are then expressible as algebraic computations inside \mathbb{F} .

4700 Representing booleans this way is convenient, because the elements 0 and 1 are defined in
 4701 any field. The representation is therefore independent of the actual field in consideration.

4702 To fix boolean algebra notation, we write 0 to represent *false* and 1 to represent *true*, and
 4703 we write \wedge to represent the boolean AND as well as \vee to represent the boolean OR operator.
 4704 The boolean NOT operator is written as \neg .

4705 **The boolean Constraint System** To represent booleans by the additive and multiplicative
 4706 neutral elements of a field, a constraint is required to actually enforce variables of boolean type
 4707 to be either 1 or 0. In fact, many of the following circuits that represent boolean functions are
 4708 only correct under the assumption that their input variables are constrained to be either 0 or 1.
 4709 Not constraining boolean variables is a common problem in circuit design.

4710 In order to constrain an arbitrary field element $x \in \mathbb{F}$ to be 1 or 0, the key observation is
 4711 that the equation $x \cdot (1 - x) = 0$ has only two solutions 0 and 1 in any field. Implementing this
 4712 equation as a circuit therefore generates the correct constraint:



4713

Using the method from XXX, we transform this circuit into the following rank-1 constraint system:

$$S_1 \cdot (1 - S_1) = 0$$

add refer-
ence

4714 Any valid assignment $\{S_1\}$ to this circuit enforces S_1 to be either 0 or 1.

4715 Some real-world circuit compilers like ZOKRATES or BELLMAN are typed, while others like
 4716 circom are not. However, all of them have their way of dealing with the binary constraint. In
 4717 PAPER, we define the following boolean type that compiles to the previous circuit:

```

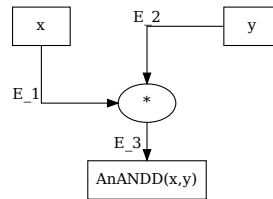
4718 type BOOL(b : BOOL) -> (x : F) {
4719     constant c1 : F = 0 ;
4720     constant c2 : F = 1 ;
4721     constant c3 : F = -1 ;
4722     c1 <== MUL( x , ADD( c2 , MUL( x , c3 ) ) ) ;
4723     x <== b ;
4724 }

```

4725 In the setup phase of a statement, we compile every occurrence of a variable of boolean type
 4726 into an instance of its associated boolean circuit.

4727 **The AND operator constraint system** Given two field elements b_1 and b_2 from \mathbb{F} that are
 4728 constrained to represent boolean variables, we want to find a circuit that computes the logical
 4729 **and** operator $AND(b_1, b_2)$ as well as its associated R1CS that enforces $b_1, b_2, AND(b_1, b_2)$ to
 4730 satisfy the constraint system if and only if $b_1 \wedge b_2 = AND(b_1, b_2)$ holds true.

4731 The key insight here is that given three boolean constraint variables b_1, b_2 and b_3 , the equa-
 4732 tion $b_1 \cdot b_2 = b_3$ is satisfied in \mathbb{F} if and only if the equation $b_1 \wedge b_2 = b_3$ is satisfied in boolean
 4733 algebra. The logical operator \wedge is therefore implementable in \mathbb{F} by field multiplication of its
 4734 arguments and the following circuit computes the \wedge operator in \mathbb{F} , assuming all inputs are re-
 4735 stricted to be 0 or 1:



4736

4737 The associated rank-1 constraint system can be deduced from the general process XXX and
 4738 consists of the following constraint

$$S_1 \cdot S_2 = S_3 \quad (7.5)$$

 add refer-
ence

4739 Common circuit languages typically provide a gadget or a function to abstract over this circuit
 4740 such that programers can use the \wedge operator without caring about the associated circuit. In
 4741 PAPER, we define the following function that compiles to the \wedge -operator's circuit:

```

4742 fn AND(b_1 : BOOL, b_2 : BOOL) -> AND(b_1, b_2) : BOOL{
4743     AND(b_1, b_2) <== MUL( b_1 , b_2 ) ;
4744 }

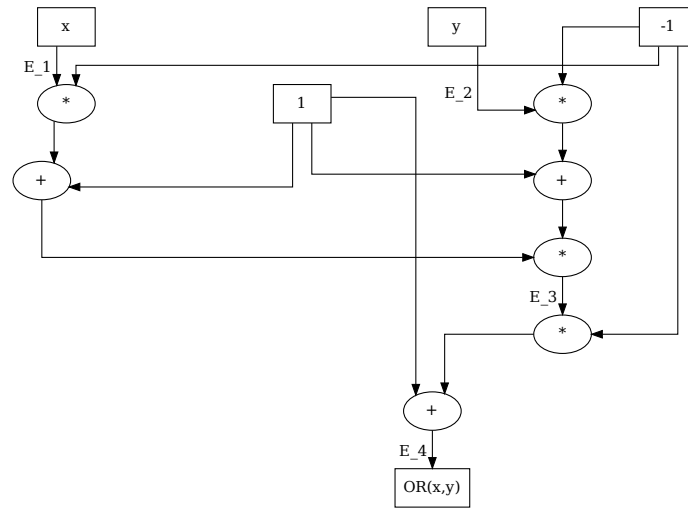
```

4745 In the setup phase of a statement, we compile every occurrence of the AND function into an
 4746 instance of its associated \wedge -operator's circuit.

4747 **The OR operator constraint system** Given two field elements b_1 and b_2 from \mathbb{F} that are
 4748 constrained to represent boolean variables, we want to find a circuit that computes the logical
 4749 **or** operator $OR(b_1, b_2)$ as well as its associated R1CS that enforces $b_1, b_2, OR(b_1, b_2)$ to satisfy
 4750 the constraint system if and only if $b_1 \vee b_2 = OR(b_1, b_2)$ holds true.

4751 Assuming that three variables b_1, b_2 and b_3 are boolean constraint, the equation $b_1 + b_2 - b_1 \cdot$
 4752 $b_2 = b_3$ is satisfied in \mathbb{F} if and only if the equation $b_1 \vee b_2 = b_3$ is satisfied in boolean algebra.
 4753 The logical operator \vee is therefore implementable in \mathbb{F} by the following circuit, assuming all
 4754 inputs are restricted to be 0 or 1:

 "constraints"
or "con-
strained"?



4755

The associated rank-1 constraint system can be deduced from the general process XXX and consists of the following constraints

add reference

$$\begin{aligned} S_1 \cdot S_2 &= S_3 \\ (S_1 + S_2 - S_3) \cdot 1 &= S_4 \end{aligned}$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the \vee operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the \vee -operator's circuit:

```
4759 fn OR(b_1 : BOOL, b_2 : BOOL) -> OR(b_1,b_2) : BOOL{
4760   constant c1 : F = -1 ;
4761   OR(b_1,b_2) <== ADD(ADD(b_1,b_2),MUL(c1,MUL(b_1,b_2))) ;
4762 }
```

In the setup phase of a statement, we compile every occurrence of the OR function into an instance of its associated \vee -operator's circuit.

Exercise 47. Let \mathbb{F} be a finite field and let b_1 as well as b_2 two boolean constraint variables from \mathbb{F} . Show that the equation $OR(b_1, b_2) = 1 - (1 - b_1) \cdot (1 - b_2)$ holds true.

Use this equation to derive an algebraic circuit with ingoing variables b_1 and b_2 and outgoing variable $OR(b_1, b_2)$ such that b_1 and b_2 are boolean **constraint** and the circuit has a valid assignment, if and only if $OR(b_1, b_2) = b_1 \vee b_2$.

"constraints" or "constrained"?

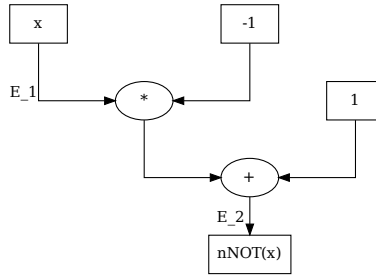
Use the technique from XXX to transform this circuit into a rank-1 constraint system and find its full solution set. Define a PAPER function that brain-compiles into the circuit.

add reference

The NOT operator constraint system Given a field element b from \mathbb{F} that is constrained to represent a boolean variable, we want to find a circuit that computes the logical NOT operator $NOT(b)$ as well as its associated RICS that enforces $b, NOT(b)$ to satisfy the constraint system if and only if $\neg b = NOT(b)$ holds true.

Assuming that two variables b_1 and b_2 are boolean **constraint**, the equation $(1 - b_1) = b_2$ is satisfied in \mathbb{F} if and only if the equation $\neg b_1 = b_2$ is satisfied in boolean algebra. The logical operator \neg is therefore implementable in \mathbb{F} by the following circuit, assuming all inputs are restricted to be 0 or 1:

"constraints" or "constrained"?



4780

The associated rank-1 constraint system can be deduced from the general process XXX and consists of the following constraints

add reference

$$(1 - S_1) \cdot 1 = S_2$$

4781 Common circuit languages typically provide a gadget or a function to abstract over this circuit
 4782 such that programers can use the \neg operator without caring about the associated circuit. In
 4783 PAPER, we define the following function that compiles to the \neg -operator's circuit:

```
4784 fn NOT(b : BOOL -> NOT(b) : BOOL{
4785   constant c1 = 1 ;
4786   constant c2 = -1 ;
4787   NOT(b_1) <== ADD( c1 , MUL( c2 , b ) ) ;
4788 }
```

4789 In the setup phase of a statement, we compile every occurrence of the NOT function into an
 4790 instance of its associated \neg -operator's circuit.

4791 *Exercise 48.* Let \mathbb{F} be a finite field. Derive the algebraic circuit and associated rank-1 constraint
 4792 system for the following operators: NOR, XOR, NAND, EQU.

4793 **Modularity** As we have seen in XXX and XXX, both algebraic circuits and R1CS have a
 4794 modularity property, and as we have seen in this section, all basic boolean functions are ex-
 4795 pressible in circuits. Combining those two properties, show that it is possible to express arbi-
 4796 trary boolean functions as algebraic circuits.

add reference

4797 This shows that the expressiveness of algebraic circuits and therefore rank-1 constraint sys-
 4798 tems is as general as the expressiveness of boolean circuits. An important implication is that the
 4799 languages $L_{R1CS-SAT}$ and $L_{Circuit-SAT}$ as defined in XXX, are as general as the famous language
 4800 L_{3-SAT} , which is known to be \mathcal{NP} -complete.

add reference

4801 *Example 129.* To give an example of how a compiler might construct complex boolean expres-
 4802 sions in algebraic circuits from simple ones and how we derive their associated rank-1 constraint
 4803 systems, let's look at the following PAPER statement:

```
4804 statement BOOLEAN_STAT {F: F_p} {
4805   fn main(b_1:BOOL,b_2:BOOL,b_3:BOOL,b_4:BOOL )-> pub b_5:BOOL {
4806     b_5 <== AND( OR( b_1 , b_2 ) , AND( b_3 , NOT( b_4 ) ) ) ;
4807   } ;
4808 }
```

add reference

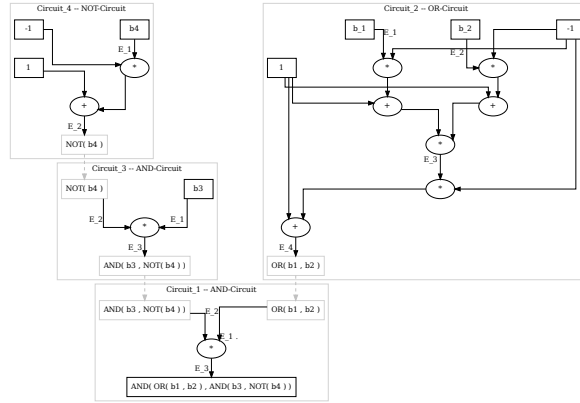
The code describes a circuit that takes four private inputs b_1, b_2, b_3 and b_4 of boolean type and computes a public output b_5 such that the following boolean expression holds true:

$$(b_1 \vee b_2) \wedge (b_3 \wedge \neg b_4) = b_5$$

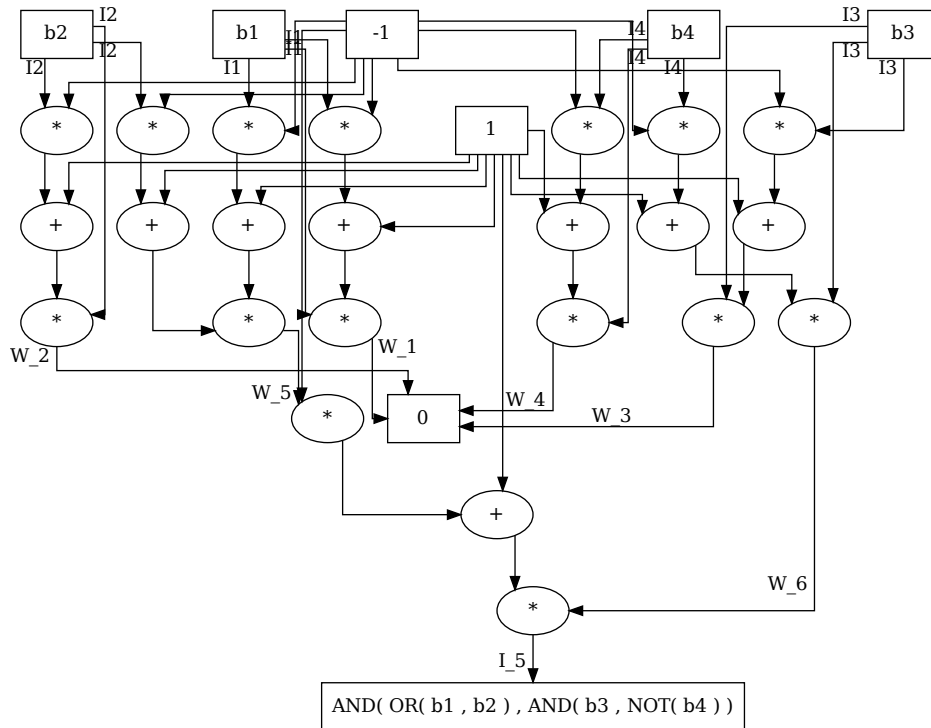
During a setup-phase, a circuit compiler transforms this high-level language statement into a circuit and associated rank-1 constraint systems and hence defines a language $L_{BOOLEAN_STAT}$.

To see how this might be achieved, we use PAPER as an example to execute the setup-phase and compile `BOOLEAN_STAT` into a circuit. Taking the definition of the boolean constraint `XXX` as well as the definitions of the appropriate boolean operators into account, we get the following circuit:

add reference



Simple optimization then collapses all box-nodes that are directly linked and all box nodes that represent the same constants. After relabeling the edges, the following circuit represents the circuit associated to the `BOOLEAN_STAT` statement:



Given some public input I_1 from \mathbb{F}_{13} , a valid assignment to this circuit consists of private inputs W_1, W_2, W_3, W_4 from \mathbb{F}_{13} such that the equation $I_1 = (W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4)$ holds true. In addition, a valid assignment also has to contain private inputs W_5, W_6, W_7, W_8, W_9 and W_{10} ,

4823 which can be derived from circuit execution. The inputs W_5, \dots, W_8 ensure that the first four
 4824 private inputs are either 0 or 1 but not any other field element, and the others enforce the boolean
 4825 operations in the expression.

To compute the associated RICS, we can use the general method from XXX and look at every labeled outgoing edge not coming from a source node. Declaring the edges coming from input nodes as well as the edge going to the single output node as public, and every other edge as private input. In this case we get:

add refer-
ence

$$\begin{aligned}
 W_5 : W_1 \cdot (1 - W_1) &= 0 && \text{boolean constraints} \\
 W_6 : W_2 \cdot (1 - W_2) &= 0 \\
 W_7 : W_3 \cdot (1 - W_3) &= 0 \\
 W_8 : W_4 \cdot (1 - W_4) &= 0 \\
 W_9 : W_1 \cdot W_2 &= W_9 && \text{first OR-operator constraint} \\
 W_{10} : W_3 \cdot (1 - W_4) &= W_{10} && \text{AND(.,NOT(.))-operator constraints} \\
 I_1 : (W_1 + W_2 - W_9) \cdot W_{10} &= I_1 && \text{AND-operator constraints}
 \end{aligned}$$

4826 The reason why this RICS only contains a single constraint for the multiplication gate in the
 4827 OR-circuit, while the general definition XXX requires two constraints, is that the second con-
 4828 straint in XXX only appears because the final addition gate is connected to an output node. In
 4829 this case, however, the final addition gate from the OR-circuit is enforced in the left factor of
 4830 the I_1 constraint. Something similar holds true for the negation circuit.

add refer-
ence

add refer-
ence

During a prover-phase, some public instance I_5 must be given. To compute a constructive proof for the statement of the associated languages with respect to instance I_5 , a prover has to find four boolean values W_1, W_2, W_3 and W_4 such that

$$(W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4) = I_5$$

4831 holds true. In our case neither the circuit nor the PAPER statement specifies how to find those
 4832 values, and it is a problem that any prover has to solve outside of the circuit. This might or
 4833 might not be true for other problems, too. In any case, once the prover found those values, they
 4834 can execute the circuit to find a valid assignment.

To give a concrete example, let $I_1 = 1$ and assume $W_1 = 1, W_2 = 0, W_3 = 1$ and $W_4 = 0$. Since $(1 \vee 0) \wedge (1 \wedge \neg 0) = 1$, those values satisfy the problem and we can use them to execute the circuit. We get

$$\begin{aligned}
 W_5 &= W_1 \cdot (1 - W_1) = 0 \\
 W_6 &= W_2 \cdot (1 - W_2) = 0 \\
 W_7 &= W_3 \cdot (1 - W_3) = 0 \\
 W_8 &= W_4 \cdot (1 - W_4) = 0 \\
 W_9 &= W_1 \cdot W_2 = 0 \\
 W_{10} &= W_3 \cdot (1 - W_4) = 1 \\
 I_1 &= (W_1 + W_2 - W_9) \cdot W_{10} = 1
 \end{aligned}$$

4835 A constructive proof of knowledge of a witness, for instance, $I_1 = 1$, is therefore given by the
 4836 tuple $P = (W_5, W_6, W_7, W_8, W_9, W_{10}) = (0, 0, 0, 0, 0, 1)$.

Arrays

The `array` type represents a fixed-size collection of elements of equal type, each selectable by one or more indices that can be computed at run time during program execution.

Arrays are a classical type, implemented by many higher programming languages that compile to circuits or rank-1 constraint systems. However, most high-level circuit languages support **static** arrays, i.e., arrays whose length is known at compile time only.

The most common way to compile arrays to circuits is to transform any array of a given type τ and size N into N circuit variables of type τ . Arrays are therefore **syntactic sugar**, that is, parts of the formal language that makes the code easier for humans to read, which the compiler transforms into input nodes, much like any other variable. In PAPER, we define the following array type:

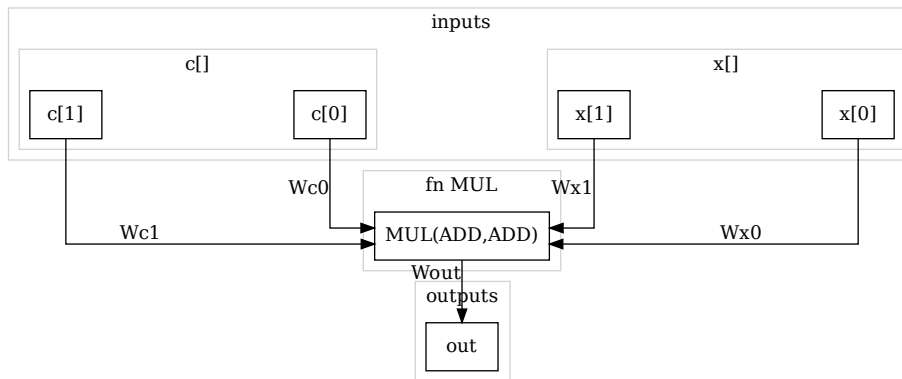
```
type <Name>: <Type>[N : unsigned] -> (Type, ...) {
  return (<Name>[0], ...)
}
```

In the setup phase of a statement, we compile every occurrence of an array of size N that contains elements of type `Type` into N variables of type `Type`.

Example 130. To give an intuition of how a real-world compiler might transform arrays into circuit variables, consider the following PAPER statement:

```
statement ARRAY_TYPE {F: F_5} {
  fn main(x: F[2]) -> F {
    let constant c: F[2] = [2, 4] ;
    let out: F <== MUL(ADD(x[1], c[0]), ADD(x[0], c[1])) ;
    return out ;
  } ;
}
```

During a setup phase, a circuit compiler might then replace any occurrence of the array type by a tuple of variables of the underlying type, and then use those variables in the circuit synthesis process instead. To see how this can be achieved, we use PAPER as an example. Abstracting over the sub-circuit of the computation, we get the following circuit:



The Unsigned Integer Type

Unsigned integers of size N , where N is usually a power of two, represent non-negative integers in the range $0 \dots 2^N - 1$. They have a notion of addition, subtraction and multiplication, defined

by modular 2^N arithmetics. If some N is given, we write uN for the associated type.

The uN Constraint System Many high-level circuit languages define the the various uN types as arrays of size N , where each element is of boolean type. This is similar to their representation on common computer hardware and allows for efficient and straightforward definition of common operators, like the various **shift**, or logical operators.

shift

If some unsigned integer N is known at compile time in PAPER, we define the following uN type:

```

4877 type uN -> BOOL[N] {
4878   let base2 : BOOL[N] <== BASE_2(uN) ;
4879   return base2 ;
4880 }
```

To enforce an N -tuple of field elements (b_0, \dots, b_{N-1}) to represent an element of type uN we therefore need N boolean constraints

$$\begin{aligned}
 S_0 \cdot (1 - S_0) &= 0 \\
 S_1 \cdot (1 - S_1) &= 0 \\
 &\dots \\
 S_{N-1} \cdot (1 - S_{N-1}) &= 0
 \end{aligned}$$

In the setup phase of a statement, we compile every occurrence of the uN type by a size N array of boolean type. During a=the prover phase, actual elements of the uN type are first transformed into binary representation and then this binary representation is assigned to the boolean array that represents the uN type.

Remark 4. Representing the uN type as boolean arrays is conceptually clean and works over generic base fields. However, representing unsigned integers in this way requires a lot of space as every bit is represented as a field element and if the base field is large, those field elements require considerable space in hardware.

It should be noted that, in some cases, there is another, more space- and constraint-efficient approach for representing unsigned integers that can be used whenever the underlying base field is sufficiently large. To understand this, recall that addition and multiplication in a prime field \mathbb{F}_p is equal to addition and multiplication of integers, as long as the sum or the product does not exceed the modulus p . It is therefore possible to represent the uN type inside the base-field type whenever N is small enough. In this case, however, care has to be taken to never overflow the modulus. It is also important to make sure that, in the case of subtraction, the subtrahend is never larger than the minuend.

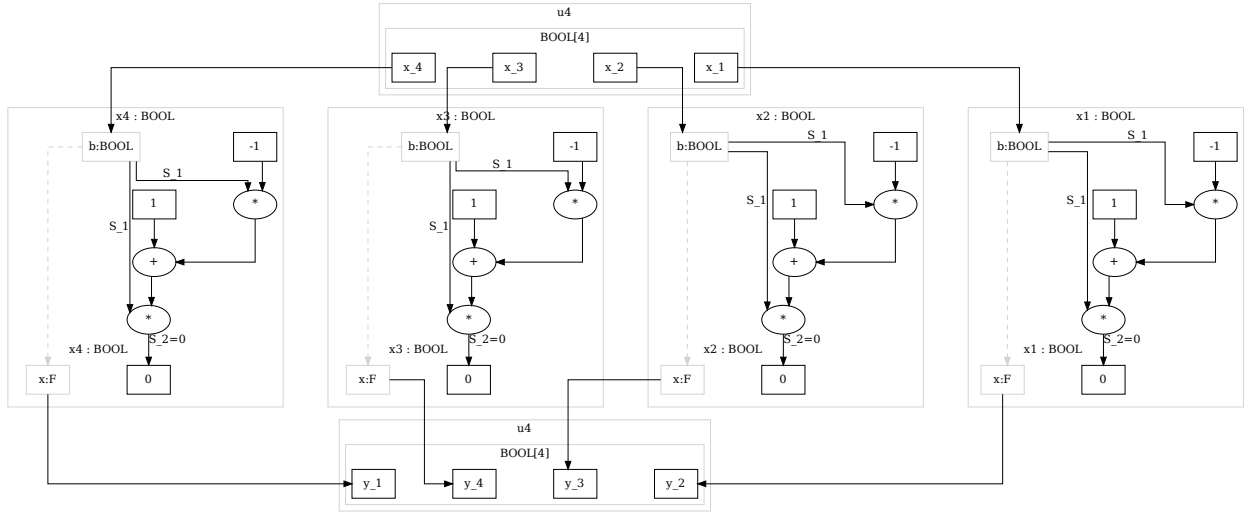
Example 131. To give an intuition of how a real-world compiler might transform unsigned integers into circuit variables, consider the following PAPER statement:

```

4899 statement RING_SHIFT{F: F_p, N=4} {
4900   fn main(x: uN)-> uN {
4901     let y:uN <== [x[1],x[2],x[3],x[0]] ;
4902     return y ;
4903   } ;
4904 }
```

During the setup-phase, a circuit compiler might then replace any occurrence of the uN type by N variables of boolean type. Using the definition of booleans, each of these variables is

then transformed into the `field` type and a boolean constraint system. To see how this can be achieved, we use PAPER as an example and get the following circuit:



During the prover phase, the function `main` is called with an actual input of `u4` type, say $x=14$. The high-level language then has to transform the decimal value 14 into its 4-bit binary representation $14_2 = (0, 1, 1, 1)$ outside of the circuit. Then the array of field values $x[4] = [0, 1, 1, 1]$ is used as an input to the circuit. Since all 4 field elements are either 0 or 1, the four boolean constraints are satisfiable and the output is an array of the four field elements $[1, 1, 1, 0]$, which represents the `u4` element 7.

The Unigned Integer Operators Since elements of `uN` type are represented as boolean arrays, shift operators are implemented in circuits simply by rewiring the boolean input variables to the output variables accordingly.

Logical operators, like AND, OR, or NOT are defined on the `uN` type by invoking the appropriate boolean operators bitwise to every bit in the boolean array that represents the `uN` element.

Addition and multiplication can be represented similarly to how machines represent those operations. Addition can be implemented by first defining the **full adder** circuit and then combining N of these circuits into a circuit that adds two elements from the `uN` type.

Exercise 49. Let $F = \mathbb{F}_{13}$ and $N=4$ be fixed. Define circuits and associated R1CS for the left and right **bishift** operators $x \ll 2$ as well as $x \gg 2$ that operate on the `uN` type. Execute the associated circuit for $x : u4 = 11$.

Exercise 50. Let $F = \mathbb{F}_{13}$ and $N=2$ be fixed. Define a circuit and associated R1CS for the addition operator $\text{ADD} : F \times F \rightarrow F$. Execute the associated circuit to compute $\text{ADD}(2, 7)$.

Exercise 51. Brain compile the following PAPER code into a circuit and derive the associated R1CS.

```
statement MASK_MERGE {F:F_5, N=4} {
  fn main(pub a : uN, pub b : uN) -> F {
    let constant mask : uN = 10 ;
    let r : uN <== XOR(a, AND(XOR(a, b), mask)) ;
    return r ;
  }
}
```

```

4938     }
4939 }

```

4940 Let L_{mask_merge} be the language defined by the circuit. Provide a constructive knowledge proof
 4941 in L_{mask_merge} for the instance $I = (I_a, I_b) = (14, 7)$.

4942 7.2.2 Control Flow

4943 Most programing languages of the imperative or functional style have some notion of basic
 4944 control structures to direct the order in which instructions are evaluated. Contemporary circuit
 4945 compilers usually provide a single thread of execution and provide basic flow constructs that
 4946 implement control flow in circuits.

4947 The Conditional Assignment

4948 Writing high-level code that compiles to circuits, it is often necessary to have a way for condi-
 4949 tional assignment of values or computational output to variables.

4950 One way to realize this in many programming languages is in terms of the conditional
 4951 ternary assignment operator $?:$ that branches the control flow of a program according to some
 4952 condition and then assigns the output of the computed branch to some variable.

```

4953 variable = condition ? value_if_true : value_if_false

```

4954 In this description, `condition` is a boolean expression and `value_if_true` as well as
 4955 `value_if_false` are expressions that evaluate to the same type as `variable`.

4956 In programming languages like Rust, another way to write the conditional assignment oper-
 4957 ator that is more familiar to many programmers is given by

```

4958 variable = if condition then {
4959     value_if_true
4960 } else {
4961     value_if_false
4962 }

```

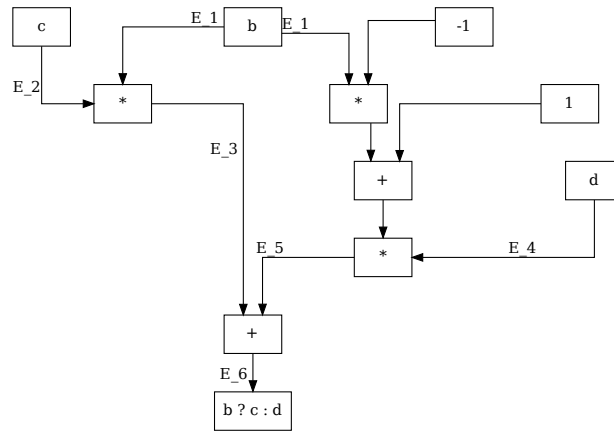
4963 In most programing languages, it is a key property of the ternary assignment operator that
 4964 the expression `value_if_true` is only evaluated if `condition` evaluates to true and the
 4965 expression `value_if_false` is only evaluated if `condition` evaluates to false. In fact,
 4966 computer programs would turn out to be very inefficient if the ternary operator would evaluate
 4967 both expressions regardless of the value of `condition`.

4968 A simple way to implement conditional assignment operator as a circuit can be achieved
 4969 if the requirement that only one branch of the conditional operator is executed is dropped. To
 4970 see that, let b , c and d be field elements such that b is a boolean constraint. In this case, the
 4971 following equation enforces a field element x to be the result of the conditional assignment
 4972 operator:

$$x = b \cdot c + (1 - b) \cdot d \quad (7.6)$$

4973 Expressing this equation in terms of the addition and multiplication operators from XXX, we
 4974 can flatten it into the following algebraic circuit:

add refer-
ence



4975

4976 Note that, in order to compute a valid assignment to this circuit, both S_2 as well as S_4 are
 4977 necessary. If the inputs to the nodes c and d are circuits themselves, both circuits need valid
 4978 assignments and therefore have to be executed. As a consequence, this implementation of
 4979 the conditional assignment operator has to execute all branches of all circuits, which is very
 4980 different from the execution of common computer programs and contributes to the increased
 4981 computational effort any prover has to invest, in contrast to the execution in other programming
 4982 models.

We can use the general technique from XXX to derive the associated rank-1 constraint system of the conditional assignment operator. We get

add reference

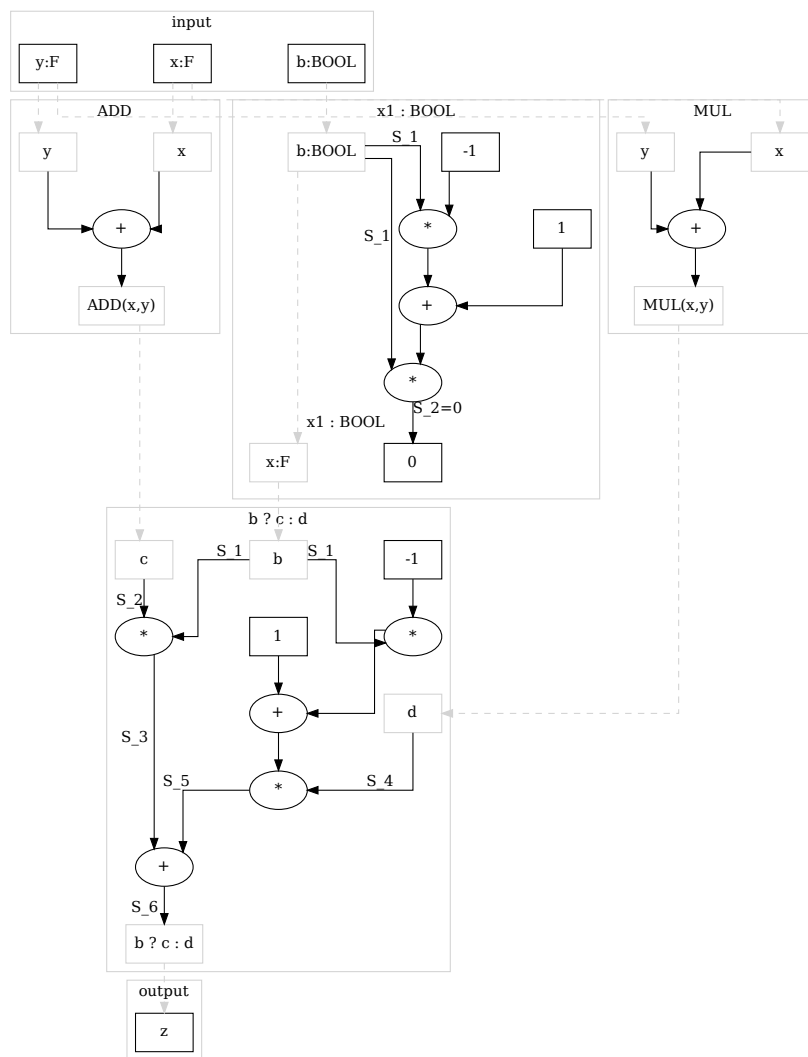
$$\begin{aligned}
 S_1 \cdot S_2 &= S_3 \\
 (1 - S_1) \cdot S_4 &= S_5 \\
 (S_3 + S_5) \cdot 1 &= S_6
 \end{aligned}$$

4983 *Example 132.* To give an intuition of how a real-world circuit compiler might transform any
 4984 high-level description of the conditional assignment operator into a circuit, consider the follow-
 4985 ing PAPER code:

```

4986 statement CONDITIONAL_OP {F:F_p} {
4987   fn main(x : F, y : F, b : BOOL) -> F {
4988     let z : F <== if b then {
4989       ADD(x,y)
4990     } else {
4991       MUL(x,y)
4992     } ;
4993     return z ;
4994   }
4995 }
```

4996 Brain compiling this code into a circuit, we first draw box nodes for all input and output vari-
 4997 ables, and then transform the boolean type into the field type together with its associated con-
 4998 straint. Then we evaluate the assignments to the output variables. Since the conditional assign-
 4999 ment operator is the top level function, we draw its circuit and then draw the circuits for both
 5000 conditional expressions. We get:



5001

5002 **Loops**

5003 In many programming languages, various loop control structures are defined that allow devel-
 5004 opers to execute expressions with a specified number of repetitions or arguments. In particular,
 5005 it is often possible to implement unbounded loops like the

5006 **while** true do { }
 5007

5008 structure, or loop structure, where the number of executions depends on execution inputs
 5009 and is therefore unknown at compile time.

5010 In contrast to this, it should be noted that algebraic circuits and rank-1 constraint systems
 5011 are not general enough to express arbitrary computation, but bounded computation only. As a
 5012 consequence, it is not possible to implement unbounded loops, or loops with bounds that are
 5013 unknown at compile time in those models. This can be easily seen since circuits are acyclic
 5014 by definition, and implementing an unbounded loop as an acyclic graph requires a circuits of
 5015 unbounded size.

5016 However, circuits are general enough to express bounded loops, where the upper bound on
 5017 its execution is known at compile time. Those loop can be implemented in circuits by enrolling
 5018 the loop.

something
missing
here?

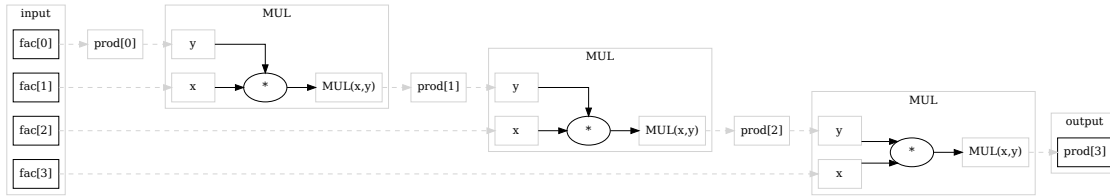
As a consequence, any programming language that compiles to algebraic circuits can only provide loop structures where the bound is a constant known at compile time. This implies that loops cannot depend on execution inputs, but on compile time parameters only.

Example 133. To give an intuition of how a real-world circuit compiler might transform any high-level description of a bounded `for` loop into a circuit, consider the following PAPER code:

```
statement FOR_LOOP {F:F_p, N: unsigned = 4} {
  fn main(fac : F[N]) -> F {
    let prod[N] : F ;
    prod[0] <== fac[0] ;
    for unsigned i in 1..N do [{
      prod[i] <== MUL(fac[i], prod[i-1]) ;
    }]
    return prod[N] ;
  }
}
```

Note that, in a program like this, the loop counter `i` has no expression in the derived circuit. It is pure syntactic **sugar**, telling the compiler how to unroll the loop.

Brain compiling this code into a circuit, we first draw box nodes for all input and output variables, noting that the loop counter is not represented in the circuit. Since all variables are of `field` type, we don't have to compile any type constraints. Then we evaluate the assignments to the output variables by unrolling the loop into 3 individual assignment operators. We get:



7.2.3 Binary Field Representations

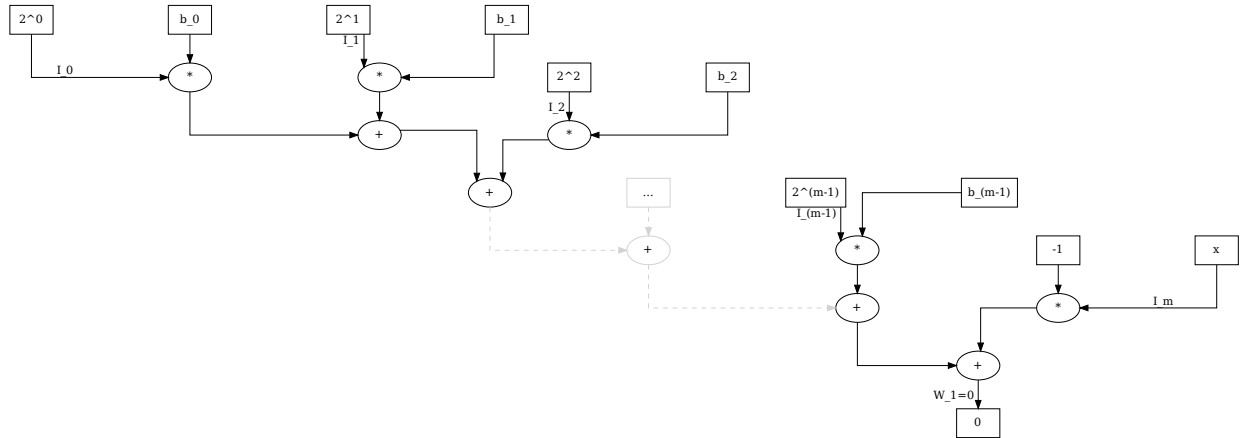
In applications, is often necessary to enforce a binary representation of elements from the `field` type. To derive an appropriate circuit over a prime field \mathbb{F}_p , let $m = |p|_2$ be the smallest number of bits necessary to represent the prime modulus p . Then a bitstring $(b_0, \dots, b_{m-1}) \in \{0, 1\}^m$ is a binary representation of a field element $x \in \mathbb{F}_p$, if and only if

$$x = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \dots + b_{m-1} \cdot 2^{m-1}$$

In this expression, addition and exponentiation is considered to be executed in \mathbb{F}_p , which is well defined since all terms 2^j for $0 \leq j < m$ are elements of \mathbb{F}_p . Note, however, that in contrast to the binary representation of unsigned integers $n \in \mathbb{N}$, this representation is not unique in general, since the modular p equivalence class might contain more than one binary representative.

Considering that the underlying prime field is fixed and the most significant bit of the prime modulus is m , the following circuit flattens equation XXX, assuming all inputs b_1, \dots, b_m are of boolean type.

5050



5051

5052

Applying the general transformation rule to compute the associated rank-1 constraint systems, we see that we actually only need a single constraint to enforce some binary representation of any field element. We get

$$(S_0 \cdot 2^0 + S_1 \cdot 2^1 + \dots + S_{m-1} \cdot 2^{m-1} - S_m) \cdot 1 = 0$$

5053 Given an array `BOOL[N]` of N boolean constraint field elements and another field element x ,
 5054 the circuit enforces `BOOL[N]` to be one of the binary representations of x . If `BOOL[N]` is not
 5055 a binary representation of x , no valid assignment and hence no solution to the associated R1CS
 5056 can exists.

5057 *Example 134.* Consider the prime field \mathbb{F}_{13} . To compute binary representations of elements
 5058 from that field, we start with the binary representation of the prime modulus 13, which is $|13|_2 =$
 5059 $(1, 0, 1, 1)$ since $13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$. So $m = 4$ and we need up to 4 bits to represent
 5060 any element $x \in \mathbb{F}_{13}$.

To see that binary representations are not unique in general, consider the element $2 \in \mathbb{F}_{13}$. It has the binary representations $|2|_2 = (0, 1, 0, 0)$ and $|2|_2 = (1, 1, 1, 1)$, since in \mathbb{F}_{13} we have

$$2 = \begin{cases} 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 \\ 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \end{cases}$$

5061 This is because the unsigned integers 2 and 15 are both in the modular 13 remainder class of 2
 5062 and hence are both representatives of 2 in \mathbb{F}_{13} .

To see how circuit XXX works, we want to enforce the binary representation of $7 \in \mathbb{F}_{13}$. Since $m = 4$ we have to enforce a 4-bit representation for 7, which is $(1, 1, 1, 0)$, since $7 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$. A valid circuit assignment is therefore given by $(S_0, S_1, S_2, S_3, S_4) = (1, 1, 1, 0, 7)$ and, indeed, the assignment satisfies the required 5 constraints including the 4 boolean constraints for S_0, \dots, S_3 :

$$\begin{aligned} 1 \cdot (1 - 1) &= 0 & // \text{boolean constraints} \\ 1 \cdot (1 - 1) &= 0 \\ 1 \cdot (1 - 1) &= 0 \\ 0 \cdot (1 - 0) &= 0 \\ (1 + 2 + 4 + 0 - 7) \cdot 1 &= 0 & // \text{binary rep. constraint} \end{aligned}$$

 add refer-
ence

7.2.4 Cryptographic Primitives

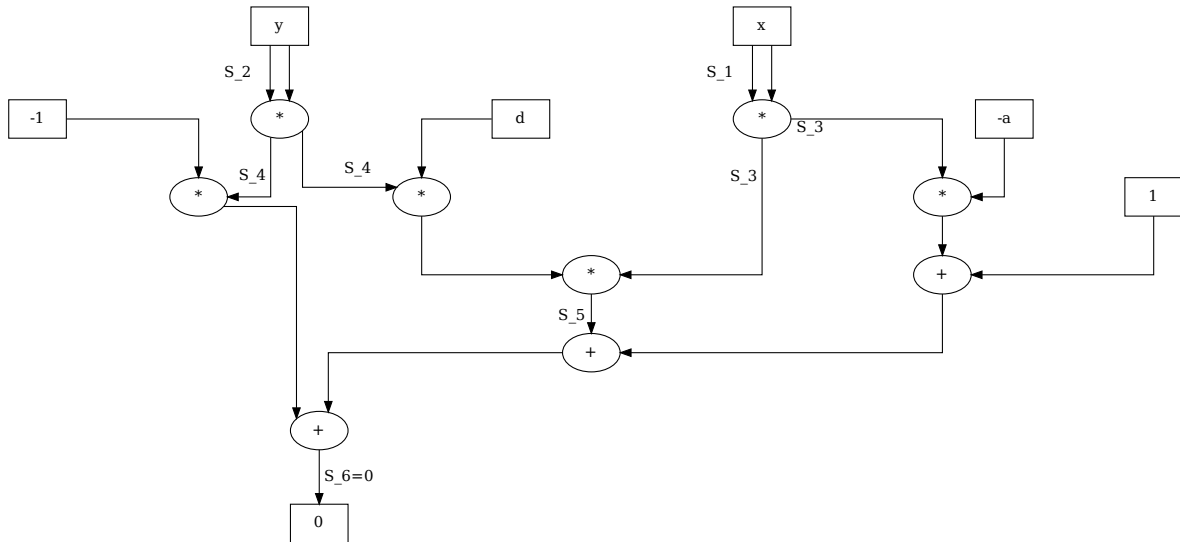
In applications, it is often required to do cryptography in a circuit. To do this, basic cryptographic primitives like hash functions or elliptic curve cryptography needs to be implemented as circuits. In this section, we give a few basic examples of how to implement such primitives.

Twisted Edwards curves

Implementing elliptic curve cryptography in circuits means to implement the field equation as well as the algebraic operations of an elliptic curve as circuits. To do this efficiently, the curve must be defined over the same base field as the field that is used in the circuit.

For efficiency reasons, it is advantageous to choose an elliptic curve such that that all required constraints and operations can be implement with as few gates as possible. Twisted Edwards curves are particularly useful for that matter, since their addition law is particularly simple and the same equation can be used for all curve points including the point at infinity. This simplifies the circuit a lot.

Twisted Edwards curves constraints As we have seen in XXX, a twisted Edwards curve over a finite field F is defined as the set of all pairs of points $(x, y) \in \mathbb{F} \times \mathbb{F}$ such that x and y satisfy the equation $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$. As we have seen in example XXX, we can transform this equation into the following circuit:



The circuit enforces the two inputs of `field` type to satisfy the twisted Edwards curve equation and, as we know from example XXX, the associated rank-1 constraint system is given by:

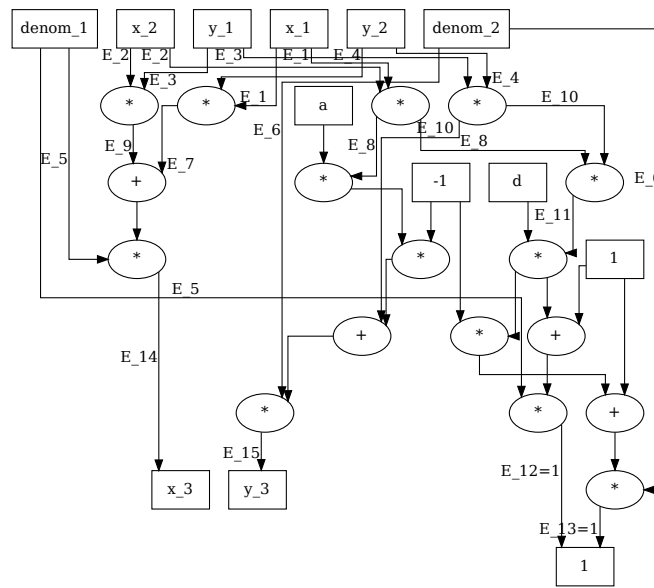
$$\begin{aligned}
 S_1 \cdot S_1 &= S_3 \\
 S_2 \cdot S_2 &= S_4 \\
 (S_4 \cdot 8) \cdot S_3 &= S_5 \\
 (12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 &= 0
 \end{aligned}$$

Exercise 52. Write the circuit and associated rank-1 constraint system for a Weierstraß curve of a given field \mathbb{F} .

Twisted Edwards curve addition As we have seen in XXX, a major advantage of twisted Edwards curves is the existence of an addition law that contains no branching and is valid for all curve points. Moreover the neutral element is not “at infinity” but the actual curve point $(0, 1)$. In fact given two points (x_1, y_1) and (x_2, y_2) on a twisted Edwards curve their sum is defined as

$$(x_3, y_3) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d \cdot x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a \cdot x_1 x_2}{1 - d \cdot x_1 x_2 y_1 y_2} \right)$$

We can use the division circuit from XXX to flatten this equation into an algebraic circuit. Inputs to the circuit are then not only the two curve points (x_1, y_1) and (x_2, y_2) , but also the two denominators $denum_1 = 1 + d \cdot x_1 x_2 y_1 y_2$ as well as $denum_2 = 1 - d \cdot x_1 x_2 y_1 y_2$, which any prover needs to compute outside of the circuit. We get



Using the general technique from XXX to derive the associated rank-1 constraint system, we get the following result:

$$\begin{aligned} S_1 \cdot S_4 &= S_7 \\ S_1 \cdot S_2 &= S_8 \\ S_2 \cdot S_3 &= S_9 \\ S_3 \cdot S_4 &= S_{10} \\ S_8 \cdot S_{10} &= S_{11} \\ S_5 \cdot (1 + d \cdot S_{11}) &= 1 \\ S_6 \cdot (1 - d \cdot S_{11}) &= 1 \\ S_5 \cdot (S_9 + S_7) &= S_{14} \\ S_6 \cdot (S_{10} - a \cdot S_8) &= S_{15} \end{aligned}$$

Exercise 53. Let \mathbb{F} be a field. Define a circuit that enforces field inversion for a point of a twisted Edwards curve over \mathbb{F} .

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>.