Lesson 9

Noir

Introduction

Noir is a domain specific language for creating and verifying proofs. Design choices are influenced heavily by Rust.

Noir is much simple and flexible in design as it does not compile immediately to a fixed NP-complete language. Instead Noir compiles to an intermediate language which itself can be compiled to an arithmetic circuit or a rank-1 constraint system.

From Documentation

Noir can be used for a variety of purposes.

Ethereum Developers

Noir currently includes a command to publish a contract which verifies your Noir program. This will be modularised in the future, however as of the alpha you can use the `contract` command to create it.

Protocol Developers

As a protocol developer, you may not want to use the Aztec backend due to it not being a fit for your stack or maybe you simply want to use a different proving system. Since Noir does not compile to a specific proof system, it is possible for protocol developers to replace the PLONK based proving system with a different proving system altogether.

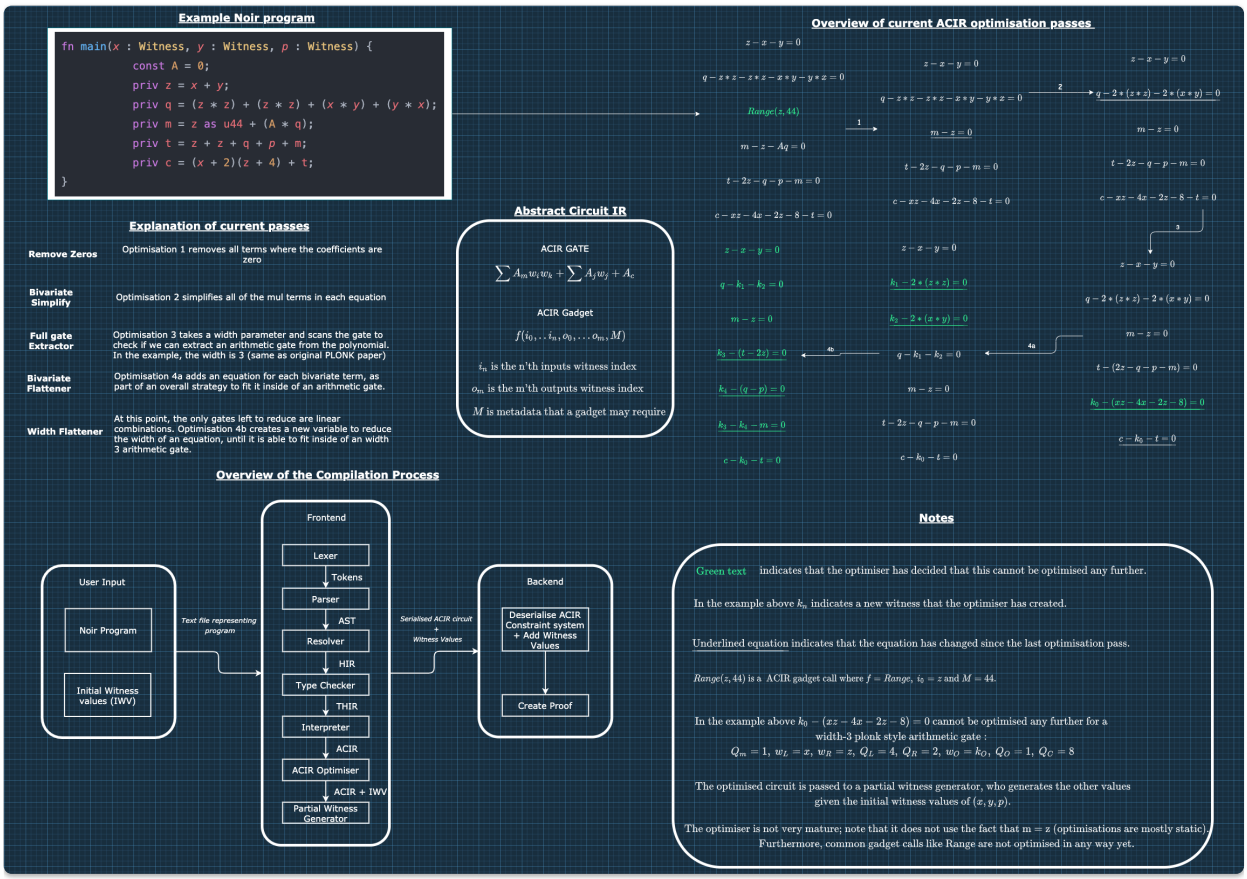
Blockchain developers

As a blockchain developer, you will be constrained by parameters set by your blockchain, ie the proving system and smart contract language has been pre-defined. In order for you to use Noir in your blockchain, a proving system backend must be implemented for it and a smart contract interface must be implemented for it.

Resources

Awesome-noir

Compilation process



Features

Backends:

- Barretenberg via FFI
- Marlin via arkworks

Compiler:

- Module System
- For expressions
- Arrays
- Bit Operations
- Binary operations `(<, <=, >, >=, +, -, *, /, %)` [See documentation for an extensive list]
- Unsigned integers
- If statements
- Structures and Tuples
- Generics

ACIR Supported OPCODES:

- Sha256
- Blake2s
- Schnorr signature verification
- Merkle Membership
- Pedersen
- HashToField

You can create a solidity verifier contract automatically

Installation

See Docs

IDEs

- Noir Editor - Browser IDE
- VSCode Extension - Syntax highlight
- Vim Plugin - Syntax highlight
- hardhat-noir - Hardhat plugin

Building the constraint system

`nargo check`

Add the inputs

In the created prover.toml add values for the inputs to the main function

Create the proof

`nargo prove p`

Verify the proof

`nargo verify p`

Language

Private & Public Types

```
fn main(x : Field, y : pub Field) -> pub Field {
    x + y
}
```

Note: Public types can only be declared through parameters on `main`.

Primitive Types

Field
Integer
Boolean

Compound Types

ARRAY

```
fn main(x : Field, y : Field) {
    let my_arr = [x, y];
    let your_arr: [Field; 2] = [x, y];
}
```

TUPLE

```
fn main() {
    let tup: (u8, u64, Field) = (255, 500, 1000);
}
```

STRUCT

```
struct Animal {
    hands: Field,
    legs: Field,
    eyes: u8,
}

fn main() {
    let legs = 4;

    let dog = Animal {
        eyes: 2,
        hands: 0,
        legs,
    };

    let zero = dog.hands;
}
```

De structuring Structs

```
fn main() {
    let Animal { hands, legs: feet, eyes } = get_octopus();

    let ten = hands + feet + eyes as u8;
}

fn get_octopus() -> Animal {
    let octopus = Animal {
        hands: 0,
```

```
        legs: 8,
        eyes: 2,
    };

    octopus
}
```

You can also define methods within a struct

```
struct MyStruct {
    foo: Field,
    bar: Field,
}

impl MyStruct {
    fn new(foo: Field) -> MyStruct {
        MyStruct {
            foo,
            bar: 2,
        }
    }

    fn sum(self) -> Field {
        self.foo + self.bar
    }
}

fn main() {
    let s = MyStruct::new(40);
    constrain s.sum() == 42;
}
```

You could also do

```
constrain MyStruct::sum(s) == 42
```

Functions

`fn` keyword

```
fn foo(x : Field, y : pub Field) -> Field {
        x + y
```

```
}
```

Loops

only for loops are possible

```
for i in 0..10 {
    // do something
};
```

Recursion is not yet possible

If expressions

```
let a = 0;
let mut x: u32 = 0;

if a == 0 {
    if a != 0 {
        x = 6;
    } else {
        x = 2;
    }
} else {
    x = 5;
    constrain x == 5;
}
constrain x == 2;
```

CONSTRAIN STATEMENT

```
fn main(x : Field, y : Field) {
    constrain x == y;
}
```

`constrain` which will explicitly constrain the predicate/comparison expression that follows to be true. If this expression is false at runtime, the program will fail to be proven.

ACIR

Noir comples to ACIR (Abstract Circuit Intermediate Representation) which later can compile to any ZK proving system.
The purpose of ACIR is to act as an intermediate layer between the proof system that Noir chooses to compile to, and the Noir syntax.
This separation between proof system and programming language, allows those who want to integrate proof systems to have a stable target.

Compiling a proof

When inside of a given Noir project the command `nargo compile my_proof` will perform two processes.

- First, compile the Noir program to its ACIR and solve the circuit's witness.
- Second, create a new `build/` directory to store the ACIR, `my_proof.acir`, and the solved witness, `my_proof.tr`

These can be used by the Noir Typescript wrapper to generate a prover and verifier inside of Typescript rather than in Nargo.

UI

https://github.com/noir-lang/noir-web-starter-next

Solidity Verifier

You can create a verifier contract for your Noir program by running:

```
nargo contract
```

A new `contract` folder would then be generated in your project directory, containing the Solidity file `plonk_vk.sol`. It can be deployed on any EVM blockchain acting as a verifier smart contract.

> *Note: It is possible to compile verifier contracts of Noir programs for other smart contract platforms as long as the proving backend supplies an implementation.*
>
> *Barretenberg, the default proving backend Nargo is integrated with, supports compilation of verifier contracts in Solidity only for the time being.*

Roadmap

Concretely the following items are on the road map:

- Prover and Verifier Key logic. (Prover and Verifier pre-process per compile)
- Fallback mechanism for backend unsupported opcodes
- Visibility modifiers

- Signed integers
- Backend integration: (Bulletproofs)
- Recursion
- Big integers

References

https://aztec.network/noir/
Developer Docs
https://docs.aztec.network/

Resources
https://github.com/noir-lang/awesome-noir

Noir Book
https://noir-lang.github.io/book/