An interesting problem with public functions which @alvaro just raised with me (having chatted with Guillaume, Maxim, Adam).

tl;dr the Public Inputs ABI needs to contain a counter

to help the kernel circuit to validate public state changes (and indeed private state changes).

Example:

contract A { // Imagine x starts life as 0: int x;

```
fn foo() {
    x++; // 0 -> 1
    B::bar();
}

fn baz() {
    x++; // 1 -> 2
}

}
```

contract B { fn bar() { A::baz(); } }

What do the state_transition

objects look like, that the Public Kernel Circuit sees, in order of processing them?

- A::foo: state_transitions = [{ slot: x, old_value: 0, new_value: 1 }, empty,...]
- B::bar: state_transitions = [empty,...]
- A: :baz state_transitions = [{ slot: x, old_value: 1, new_value: 2 }, empty,...]

At the moment, our approach is to ‘squash’ state transitions for x

into a single state transition on the Public Kernel Circuit’s own

state_transitions

array. So, ultimately, it would be squashed to:

- `kernel_state_transitions = [{ slot: x, old_value: 0, new_value: 2 }, empty,…]

BUT WHAT IF WE MODIFIED A::foo

TO CALL B::bar

FIRST, before incrementing x

:

contract A { // Imagine x starts life as 0: int x;

```
fn foo() {
    B::bar();
    x++; // 1 -> 2
}

fn baz() {
    x++; // 0 -> 1
}

}
```

contract B { fn bar() { A::baz(); } }

Notice now the ordering of the transitions has been switched. But the Kernel Circuit has no information to be able to figure out which ordering to use!

Repeating the above… What do the state_transition

objects look like, that the Public Kernel Circuit sees, in order of processing them?

- A::foo: state_transitions = [{ slot: x, old_value: 1, new_value: 2 }, empty,...]

<— NO LONGER THE FIRST TRANSITION

- B::bar: state_transitions = [empty,...]
- A: :baz state_transitions = [{ slot: x, old_value: 0, new_value: 1 }, empty,...]

Ultimately, this SHOULD be squashed to:

- `kernel_state_transitions = [{ slot: x, old_value: 0, new_value: 2 }, empty,…]

… just like the first example. But the Kernel Circuit doesn't know which transition happened first!

Suggestion to fix this problem:

Each function also tracks a counter

, and (I'm open to suggestions for names

). Maybe a clause

counter, because it tracks clauses

of the body of each function.

fn foo () { // counter = 0 bar() { // counter++ // = 1 baz() { // counter++ // = 2 } // counter++ // = 3 } // counter++ // = 4 }

Each function's Public Inputs ABI then would need to expose its start_counter

and end_counter

. These values would go nicely inside the call_context

of the ABI. Our Noir Contract standard libraries (which Alvaro has been writing) can correctly track these counters, every time a call

is made).

Any state_reads

which happen when the counter is n

, would need to contain this extra piece of information:

state_read = { counter, storage_slot, value }

, and similarly:

state_transitions = { counter, storage_slot, old_value, new_value }

Now if we repeat our 2nd example:

- A::foo: state_transitions = [{ counter: 4, slot: x, old_value: 1, new_value: 2 }, empty,...]
- B::bar: state_transitions = [empty,...]
- A: :baz state_transitions = [{ counter: 2, slot: x, old_value: 0, new_value: 1 }, empty,...]

Great! The Kernel Circuit can now see that the final-listed transition of x

was the first to occur. So it can double-check that the new_value = 1

of the first-ordered transition of x

(from 0 → 1) matches the old_value = 1

of the next-ordered transition of x

(from 1 → 2).