The goal of these posts is to guide you through the main components involved in an application in Taiga and to create an application from scratch.

Notes, applications and validity predicates

Validity predicate

What is a validity predicate? The name seems to indicate that it is a boolean function (predicate) that checks that something

is valid. So, what is valid

in a validity predicate? In the most general sense, predicates validate transactions. But different predicates validate different necessary parts for a transaction to be validated. In particular, every note comes with an application

validity predicate, that is, a predicate that validates an application. The part of the application validity predicate that is shared among users (i.e. the rules of an application that is independent of who owns

the application) is called static

. The part of an application that may vary from user to user when they use the application, such as a signature scheme, is called dynamic

.

At a lower level of abstraction, a validity predicate is a circuit that holds certain additional properties, namely properties related with the notes they are contained in.

Taiga is using Halo2 as its backend or proving system. In this post we will say circuit to mean a Halo2 circuit, or, more specifically, a Plonkish arithmetization, i.e. the language that expresses circuits for the Halo2 proving system in which circuits are defined in terms of a rectangular matrix of values.

pub trait ValidityPredicateCircuit: Circuit + ValidityPredicateInfo + ValidityPredicateVerifyingInfo { type VPConfig: ValidityPredicateConfig + Clone;

```
fn basic_constraints(
    &self,
    config: Self::VPConfig,
    mut layouter: impl Layouter<pallas::Base>,
) -> Result<BasicValidityPredicateVariables, Error> {
    // Default implementation, constrains the notes integrity.
    // ...
    // (Some code comes here)
    // ...
    Ok(BasicValidityPredicateVariables {
        owned_note_pub_id,
        input_note_variables: input_note_variables.try_into().unwrap(),
        output_note_variables: output_note_variables.try_into().unwrap(),
    })
}

fn custom_constraints(
    &self,
    _config: Self::VPConfig,
    mut _layouter: impl Layouter<pallas::Base>,
    _basic_variables: BasicValidityPredicateVariables,
) -> Result<(), Error>

}
```

At first glance, we notice that a validity predicate requires the implementation of two methods: basic_constraints

and custom_constraints

, and setting a configuration type VPConfig

. It extends other traits: Circuit

, ValidityPredicateInfo

and ValidityPredicateVerifyingInfo

. We'll have a quick look at these next.

Basic constraints

These are the constraints that most validity predicates are likely to have, independently of the application. They check the integrity

of the notes, that is, that the given public note commitments and nullifiers were computed correctly for some hidden given notes. How do we do this?

We first need to make these nullifiers and commitments available to the circuit as public inputs. As we noticed in the snippet above, the trait ValidityPredicateCircuit

also extends ValidityPredicateInfo

. The default implementation of get_mandatory_public_inputs

in ValidityPredicateInfo

does exactly this, that is, it adds the input note nullifiers and the output note commitments to the public inputs of the validity predicate in a particular order.

pub trait ValidityPredicateInfo { fn get_input_notes(&self) -> &[Note; 2]; fn get_output_notes(&self) -> &[Note; 2]; fn get_mandatory_public_inputs(&self) -> Vec { let mut public_inputs = vec![]; self.get_input_notes() .iter() .zip(self.get_output_notes().iter()) .for_each(|(input_note, output_note)| { let nf = input_note.get_nf().unwrap().inner(); public_inputs.push(nf); let cm = output_note.commitment(); public_inputs.push(cm.get_x()); }); public_inputs.push(self.get_owned_note_pub_id()); public_inputs } fn get_public_inputs(&self, rng: impl RngCore) -> ValidityPredicatePublicInputs; fn get_owned_note_pub_id(&self) -> pallas::Base; }

In the context of Taiga, a validity predicate is instantiated in one of the four notes that are referenced in a partial transaction. Roughly, a partial transaction consists of two notes that are consumed and two notes that are created. How does a validity predicate know which of these four notes it belongs to? And why is it important that a validity predicate knows which of the notes they own? The identifier

of a note is either the nullifier of the note if it is an input note or the commitment of the note if it is an output note. This identifier is the key to look up the target variables and determine whether the owned note is an input or an output note. Knowing which note the validity predicate owns is necessary to, for example, encrypt the note with the verifier's public key or to constrain the dynamic validity predicates that are encoded in app_data_dynamic

, as we'll see later. In our implementation above, the identifier of a note is appended to each validity predicate public inputs.

The role of the basic_constraints

method is to reconstruct the note commitments and nullifiers in the circuit

with the note private inputs and check these match the public inputs (also called instances), that is, the commitments and nullifiers.

layouter.constrain_instance(nf.cell(), instances, nf_row_idx)?;

layouter.constrain_instance(cm_x.cell(), instances, cm_row_idx)?;

Since instances

in a Halo2 circuit is an array of finite fields, nf_row_idx

and cm_row_idx

are just pointing to the right index of the instances or public inputs. That is, we are checking that the nullifier value nf

we have constructed in the circuit using note data as private inputs is equal to the nf_row_idx

entry of the public inputs.

Lastly, a validity predicate is required to provide a way to create the information for a verifier to check the validity of the application logic, that is, its proof

, its public inputs

and its verifying key

. In other words, a validity predicate must determine how to prove itself via the get_verifying_info

method.

pub trait ValidityPredicateVerifyingInfo: Circuit { fn get_verifying_info(&self) -> VPVerifyingInfo { let mut rng = OsRng; let params = SETUP_PARAMS_MAP.get(&12).unwrap(); let vk = keygen_vk(params, self).expect("keygen_vk should not fail");

let pk = keygen_pk(params, vk.clone(), self).expect("keygen_pk should not fail"); let public_inputs = self.get_public_inputs(&mut rng); let proof = Proof::create(&pk, params, self.clone(), &[public_inputs.inner()], &mut rng,) .unwrap(); VPVerifyingInfo { vk, proof, public_inputs, } }

```
fn get_vp_vk(&self) -> ValidityPredicateVerifyingKey {
    let params = SETUP_PARAMS_MAP.get(&12).unwrap();
    let vk = keygen_vk(params, self).expect("keygen_vk should not fail");
    ValidityPredicateVerifyingKey::from_vk(vk)
}
```

}

Custom constraints

The custom contraints

method can be seen as the part of the Halo2 Circuit

’s synthesize

method that deals with the logic that makes each validity predicate unique. What is shared among validity predicates is hidden under basic_constraints

. The basic_constraints

method returns a context

consisting of information about the input and output notes that is then available for custom_constraints

. This context is called BasicValidityPredicateVariables

:

pub struct BasicValidityPredicateVariables { pub owned_note_pub_id: AssignedCell, pub input_note_variables: [InputNoteVariables; NUM_NOTE], pub output_note_variables: [OutputNoteVariables; NUM_NOTE], }

As mentioned, a validity predicate also extends the trait Circuit

from the Halo2 library, so an implementation of this trait is also required. For our purposes, this trait looks roughly as follows:

pub trait Circuit { // ... fn configure(meta: &mut ConstraintSystem) -> Self::Config;

fn synthesize(&self, config: Self::Config, layouter: impl Layouter<F>) -> Result<(), Error>;

}

The two main methods of the Circuit

trait are configure

and synthesize

. The configure

method must describe the exact gate arrangement, column arrangement, etc. of the circuit. The synthesize

method contains the application logic of the circuit, i.e. given the configuration matrix

, it describes the circuit constraints. Taiga conceptually divides the synthesize

method into basic and custom constraints, and it gives a default implementation for the basic constraints. This allows the user of Taiga to focus on implementing what is relevant for their specific application, hiding note and other implementation details.

The vp_circuit_impl!

macro generates a Circuit

implementation for validity predicate. Taiga’s philosophy is to maximise expressibility and minimise compromises. In this case, the user is also able to implement the Circuit

trait in a different way.

// From the vp_circuit_impl! macro impl Circuitfor $name { // ... some code here fn configure(meta: &mut ConstraintSystem)

-> Self::Config { Self::Config::configure(meta) } fn synthesize(&self, config: Self::Config, mut layouter: impl Layouter,) -> Result<(), Error> { let basic_variables = self.basic_constraints(config.clone(), layouter.namespace(|| "basic constraints"),)?; self.custom_constraints(config, layouter.namespace(|| "custom constraints"), basic_variables,)?; Ok(()) } }

Notice how the basic variables derived from the basic_constraints

method feed into the custom constraints in the synthesize

method. This means that the information of all input and output notes are also accessible in custom_constraints

.

Configuration

We placed the shared logic among validity predicates (i.e. the note integrity checks) in the basic_constraints

method while custom_constraints

contains the application logic. Similarly, the configuration of a validity predicate separates the note-specific configuration from the application-specific configuration, hiding the former from the user with a default implementation.

pub trait ValidityPredicateConfig { fn configure_note(meta: &mut ConstraintSystem) -> NoteConfig { let instances = meta.instance_column(); meta.enable_equality(instances);

```
    let advices = [
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
    ];

    for advice in advices.iter() {
        meta.enable_equality(*advice);
    }

    NoteChip::configure(meta, instances, advices)
}
fn get_note_config(&self) -> NoteConfig;
fn configure(meta: &mut ConstraintSystem<pallas::Base>) -> Self;

}
```

What are these advice columns? Why do we need so many of them? In the configuration phase, we are basically constructing a matrix representation of the circuit.

Advice 0

Advice 1

…

Advice 9

Instance 0

Fixed 0

…

Fixed n

input 0

input 1

…

input 9

output

s_{00}

…

s_{n0}

a_{01}

a_{11}

…

a_{91}

s_{01}

…

s_{n0}

a_{02}

a_{12}

…

a_{92}

s_{02}

…

s_{n0}

…

…

…

…

…

…

…

…

a_{0m}

a_{1m}

…

a_{9m}

s_{0m}

…

s_{n0}

Advice columns correspond to witness values. In this case, we are allocating ten columns to the note chip configuration. There are many operations in this note chip that require allocating many values in different cells, such as a poseidon hash, elliptic curve operations, note commitments, etc. Using more columns or more rows for this is a design choice and has performance implications in both prover and verifier.

Fixed columns are fixed by the circuit. They are usually selectors. Selectors are boolean values that help us create custom gates. For example, in the vanilla Plonk equation

$$a(X)b(X)q\_M(X)+a(X)q\_L(X)+b(X)q\_R(X)+c(X)q\_O(X)+PI(X)+q\_C(X)=0$$

all q_M

, q_L

, q_R

, q_O

, and q_C

are selectors. If, say, only q_M

is set to 1 and the other selectors are set to 0, the gate becomes a multiplication gate.

Lastly, instance columns are used for public inputs. The prover will fill the cells in the advice and instance columns during proof generation.

Each specific gate in a circuit requires its own configuration, it be an addition gate or a gate to get the note that the validity predicate owns. Since most validity predicates will share these gates, Taiga also provides a “general” configuration for validity predicates.

[derive(Clone, Debug)]

pub struct GeneralVerificationValidityPredicateConfig { pub note_config: NoteConfig, pub advices: [Column; 10], pub instances: Column, pub get_is_input_note_flag_config: GetIsInputNoteFlagConfig, pub get_owned_note_variable_config: GetOwnedNoteVariableConfig, pub conditional_equal_config: ConditionalEqualConfig, pub extended_or_relation_config: ExtendedOrRelationConfig, pub add_config: AddConfig, pub sub_config: SubConfig, pub mul_config: MulConfig, }

Note

As explained in Anatomy of a partial transaction: Part 1:

A note represents a subset of the state of the system. Every note is characterised by the verifying key vk

(i.e. a succint representation of the validity predicate) and the encoded data in app_data_static

. These two fields constitute the type of the note. The note type is a reference to what we commonly call application

.

Application

An application in Taiga is then a program uniquely determined by its logic and its (static) data. For example, two different tokens are two different applications and a Sudoku game is an application, too. Since a token is a well-understood type of application, we’ll take it as example in our next post.

Summary

We have briefly covered the inner workings of a validity predicate and what an application in Taiga is. Next, we’ll create a token application.