API

Structs

`Maybe`

Encapsulates an optionally present value. If `isSome` is `false`, `value` should be `default<T>` by convention.

```
struct Maybe<T> {
  isSome: Boolean;
  value: T;
}
```

`Either`

Disjoint union of `A` and `B`. Iff `isLeft` if `true`, `left` should be populated, otherwise `right`. The other should be `default<` Ask AI
convention.

```
struct Either<A, B> {
  isLeft: Boolean;
```

Feedback

```
  left: A;
  right: B;
}
```

`CurvePoint`

A point on the proof systems embedded curve, in affine coordinates.

Only outputs of elliptic curve operations are actually guaranteed to lie on the curve.

```
struct CurvePoint {
  x: Field;
  y: Field;
}
```

`MerkleTreeDigest`

The root hash of a Merkle tree, represented by a single `Field`.

```
struct MerkleTreeDigest { field: Field; }
```

`MerkleTreePathEntry`

An entry in a Merkle tree path, indicating if the path leads left or right, and the root of the sibling node. Primarily used in `MerkleTreePath`

Feedback

```
struct MerkleTreePathEntry {
  sibling: MerkleTreeDigest;
  goesLeft: Boolean;
}
```

`MerkleTreePath`

A path in a depth `n` Merkle tree, leading to a leaf of type `T`. Primarily used for `merkleTreePathRoot`.

This can be constructed from `witness`es that use the compiler output's `findPathForLeaf` and `pathForLeaf` functions.

```
struct MerkleTreePath<#n, T> {
  leaf: T;
  path: Vector<n, MerkleTreePathEntry>;
}
```

`ContractAddress`

The address of a contract, used as a recipient in `send`, `sendImmediate`, `createZswapOutput`, and `mintToken`.

```
struct ContractAddress { bytes: Bytes<32>; }
```

`CoinInfo`

The description of a newly created shielded coin, used in outputting shielded coins, or spending/receiving shielded coins t
the current transaction.

Feedback

`nonce` can be deterministically derived with `evolveNonce`.

Used in:

- `receive`
- `sendImmediate`
- `mergeCoin`
- `mergeCoinImmediate`
- `createZswapOutput`

```
struct CoinInfo {
  nonce: Bytes<32>;
  color: Bytes<32>;
  value: Uint<128>;
}
```

`QualifiedCoinInfo`

The description of an existing shielded coin in the ledger, ready to be spent.

Used in:

- `send`
- `mergeCoin`
- `mergeCoinImmediate`
- `createZswapInput`

Feedback

```
struct QualifiedCoinInfo {
  nonce: Bytes<32>;
  color: Bytes<32>;
  value: Uint<128>;
  mtIndex: Uint<64>;
}
```

`ZswapCoinPublicKey`

The public key used to output a `CoinInfo` to a user, used as a recipient in `send`, `sendImmediate`, and `createZswapOutput`.

```
struct ZswapCoinPublicKey { bytes: Bytes<32>; }
```

`SendResult`

The output of `send` and `sendImmediate`, detailing the created shielded coin, and the change from spending the input, if applicable.

```
struct SendResult {
  change: Maybe<CoinInfo>;
  sent: CoinInfo;
}
```

Circuits

`some`

Feedback

Constructs a `Maybe<T>` containing an element of type `T`

```
circuit some<T>(value: T): Maybe<T>;
```

`none`

Constructs a `Maybe<T>` containing nothing

```
circuit none<T>(): Maybe<T>;
```

`left`

Construct an `Either<A, B>` containing the `A` item of the disjoint union

```
circuit left<A, B>(value: A): Either<A, B>;
```

`right`

Constructs an `Either<A, B>` containing the `B` item of the disjoint union

```
circuit right<A, B>(value: B): Either<A, B>;
```

`transientHash`

Feedback

Builtin transient hash compression function

This function is a circuit-efficient compression function from arbitrary values to field elements, which is not guaranteed to persist between upgrades. It should not be used to derive state data, but can be used for consistency checks.

Although this function returns a hash of its inputs, it is not considered sufficient to protect its input from disclosure. If its input contains any value returned from a witness, the program must acknowledge disclosure (via a `disclose` wrapper) if the result can be stored in the public ledger, returned from an exported circuit, or passed to another contract via a cross-contract call.

```
circuit transientHash<T>(value: T): Field;
```

`transientCommit`

Builtin transient commitment function

This function is a circuit-efficient commitment function over arbitrary types, and a field element commitment opening, to field elements, which is not guaranteed to persist between upgrades. It should not be used to derive state data, but can be used for consistency checks.

Unlike `transientHash`, this function is considered sufficient to protect its input from disclosure, under the assumption that the `rand` argument is sufficiently random. Thus, even if its input contains a value or values returned from one or more witnesses, the program need not acknowledge disclosure (via a `disclose` wrapper) if the result can be stored in the public ledger, returned from an exported circuit, or passed to another contract via a cross-contract call.

```
circuit transientCommit<T>(value: T, rand: Field): Field;
```

Feedback

`persistentHash`

Builtin persistent hash compression function

This function is a non-circuit-optimised compression function from arbitrary values to a 256-bit bytestring. It is guaranteed to persist between upgrades, and to consistently use the SHA-256 compression algorithm. It *should* be used to derive state data, and not for consistency checks where avoidable.

The note about disclosing under `transientHash` also applies to this function.

```
circuit persistentHash<T>(value: T): Bytes<32>;
```

`persistentCommit`

Builtin persistent commitment function

This function is a non-circuit-optimised commitment function from arbitrary values representable in Compact, and a 256-bit bytestring opening, to a 256-bit bytestring. It is guaranteed to persist between upgrades, and use the SHA-256 compression algorithm. It *should* be used to derive state data, and not for consistency checks where avoidable.

The note about disclosing under `transientCommit` also applies to this function.

```
circuit persistentCommit<T>(value: T, rand: Bytes<32>): Bytes<32>;
```

`degradeToTransient`

Feedback

This function "degrades" the output of a `persistentHash` or `persistentCommit` to a field element, which can then be used in `transientHash` or `transientCommit`.

```
circuit degradeToTransient(x: Bytes<32>) : Field;
```

`upgradeFromTransient`

This function "upgrades" a field element to the output of a `persistentHash` or `persistentCommit`.

```
circuit upgradeFromTransient(x: Field): Bytes<32>;
```

`ecAdd`

This function add two elliptic `CurvePoint`s (in multiplicative notation)

```
circuit ecAdd(a: CurvePoint, b: CurvePoint): CurvePoint;
```

`ecMul`

This function multiplies an elliptic `CurvePoint` by a scalar (in multiplicative notation)

```
circuit ecMul(a: CurvePoint, b: Field): CurvePoint;
```

Feedback

`ecMulGenerator`

This function multiplies the primary group generator of the embedded curve by a scalar (in multiplicative notation)

```
circuit ecMulGenerator(b: Field): CurvePoint;
```

`hashToCurve`

This function maps arbitrary types to `CurvePoint`s.

Outputs are guaranteed to have unknown discrete logarithm with respect to the group base, and any other output, but are not guaranteed to be unique (a given input can be proven correct for multiple outputs).

Inputs of different types `T` may have the same output, if they have the same field-aligned binary representation.

```
circuit hashToCurve<T>(value: T): CurvePoint;
```

`merkleTreePathRoot`

Derives the Merkle tree root of a `MerkleTreePath`, which should match the root of the tree that this path originated from.

```
circuit merkleTreePathRoot<#n, T>(path: MerkleTreePath<n, T>): MerkleTreeDigest;
```

`merkleTreePathRootNoLeafHash`

Feedback

Derives the Merkle tree root of a `MerkleTreePath`, which should match the root of the tree that this path originated from. As opposed to `merkleTreePathRoot`, this variant assumes that the tree leaves have already been hashed externally.

```
circuit merkleTreePathRootNoLeafHash<#n>(path: MerkleTreePath<n, Bytes<32>>): MerkleTreeDigest;
```

`nativeToken`

Returns the token type of the native token

```
circuit nativeToken(): Bytes<32>;
```

`tokenType`

Transforms a domain separator for the given contract into a globally namespaced token type. A contract can issue tokens for its domain separators, which lets it create new tokens, but due to collision resistance, it cannot mint tokens for another contract's token type. This is used as the `color` field in `CoinInfo`.

```
circuit tokenType(domainSep: Bytes<32>, contract: ContractAddress): Bytes<32>;
```

`mintToken`

Creates a new shielded coin, minted by this contract, and sends it to the given recipient. Returns the corresponding `CoinInfo`. This requires inputting a unique nonce to function securely, it is left to the user how to produce this.

Feedback

```
circuit mintToken(
  domainSep: Bytes<32>,
  value: Uint<128>,
  nonce: Bytes<32>,
  recipient: Either<ZswapCoinPublicKey, ContractAddress>
): CoinInfo;
```

evolveNonce

Deterministically derives a `CoinInfo` nonce from a counter index, and a prior nonce.

```
circuit evolveNonce(
  index: Uint<64>,
  nonce: Bytes<32>
): Bytes<32>;
```

burnAddress

Returns a payment address that guarantees any shielded coins sent to it are burned.

```
circuit burnAddress(): Either<ZswapCoinPublicKey, ContractAddress>;
```

receive

Receives a shielded coin, adding a validation condition requiring this coin to be present as an output addressed to this contract, and not received by another call

```
circuit receive(coin: CoinInfo): [];
```

`send`

Sends given value from a shielded coin owned by the contract to a recipient. Any change is returned and should be managed by the contract.

Note that this does not currently create coin ciphertexts, so sending to a user public key except for the current user will not lead to this user being informed of the coin they've been sent.

```
circuit send(input: QualifiedCoinInfo, recipient: Either<ZswapCoinPublicKey, ContractAddress>, valu
e: Uint<128>): SendResult;
```

`sendImmediate`

Like `send`, but for coins created within this transaction

```
circuit sendImmediate(input: CoinInfo, target: Either<ZswapCoinPublicKey, ContractAddress>, value:
Uint<128>): SendResult;
```

`mergeCoin`

Takes two coins stored on the ledger, and combines them into one

Feedback

```
circuit mergeCoin(a: QualifiedCoinInfo, b: QualifiedCoinInfo): CoinInfo;
```

`mergeCoinImmediate`

Takes one coin stored on the ledger, and one created within this transaction, and combines them into one

```
circuit mergeCoinImmediate(a: QualifiedCoinInfo, b: CoinInfo): CoinInfo;
```

`ownPublicKey`

Returns the `ZswapCoinPublicKey` of the end-user creating this transaction.

```
circuit ownPublicKey(): ZswapCoinPublicKey;
```

`createZswapInput`

Notifies the context to create a new Zswap input originating from this call. Should typically not be called manually, prefer `send` and `sendImmediate` instead.

The note about disclosing under `transientHash` also applies to this function.

```
circuit createZswapInput(coin: QualifiedCoinInfo): [];
```

Feedback

`createZswapOutput`

Notifies the context to create a new Zswap output originating from this call. Should typically not be called manually, prefer `send` and `sendImmediate`, and `receive` instead.

The note about disclosing under `transientHash` also applies to this function.

```
circuit createZswapOutput(coin: CoinInfo, recipient: Either<ZswapCoinPublicKey, ContractAddress>):
[];
```

`blockTimeLt`

Returns true if the current block time is less than the given value.

```
circuit blockTimeLt(time: Uint<64>): Boolean;
```

`blockTimeGte`

Returns true if the current block time is greater than or equal to the given value.

```
circuit blockTimeGte(time: Uint<64>): Boolean;
```

`blockTimeGt`

Returns true if the current block time is greater than the given value.

Feedback

```
circuit blockTimeGt(time: Uint<64>): Boolean;
```

`blockTimeLte`

Returns true if the current block time is less than or equal to the given value.

```
circuit blockTimeLte(time: Uint<64>): Boolean;
```

Feedback