ANASTASIA LABS

Security Audit Report

OpShin Audit

Date June, 2025
Project OpShin Audit
Version 1.0

Contents

Disclosure

This document contains proprietary information belonging to Anastasia Labs. Duplication, redistribution, or use, in whole or in part, in any form, requires explicit consent from Anastasia Labs.

Nonetheless, both the customer **OpShin** and Anastasia Labs are authorized to share this document with the public to demonstrate security compliance and transparency regarding the outcomes of the Protocol.

Disclaimer and Scope

A code review represents a snapshot in time, and the findings and recommendations presented in this report reflect the information gathered during the assessment period. It is important to note that any modifications made outside of this timeframe will not be captured in this report.

While diligent efforts have been made to uncover potential vulnerabilities, it is essential to recognize that this assessment may not uncover all potential security issues in the protocol.

It is imperative to understand that the findings and recommendations provided in this audit report should not be construed as investment advice.

Furthermore, it is strongly recommended that projects consider undergoing multiple independent audits and/or participating in bug bounty programs to increase their protocol security.

Please be aware that the scope of this security audit does not extend to the compiler layer, such as the UPLC code generated by the compiler or any areas beyond the audited code.

The scope of the audit did not include additional creation of unit testing or property-based testing of the contracts.

Assessment Overview

From **October 21**, **2024** to **May 5**, **2025**, **OpShin** engaged Anastasia Labs to evaluate and conduct a security assessment of its **Opshin** codebase. All code revision was performed following industry best practices.

Phases of code auditing activities include the following:

- **Planning** – Customer goals are gathered.
- **Discovery** – Perform code review to identify potential vulnerabilities, weak areas, and exploits.
- **Attack** – Confirm potential vulnerabilities through testing and perform additional discovery upon new access.
- **Reporting** – Document all found vulnerabilities.

The engineering team has also conducted a comprehensive review of protocol optimization strategies.

Each issue was logged and labeled with its corresponding severity level, making it easier for our audit team to manage and tackle each vulnerability.

Assessment Components

Manual Revision

Our manual code auditing is focused on a wide range of attack vectors, including but not limited to:

- UTXO Value Size Spam (Token Dust Attack)
- Large Datum or Unbounded Protocol Datum
- EUTXO Concurrency DoS
- Unauthorized Data modification
- Multisig PK Attack
- Infinite Mint
- Incorrect Parameterized Scripts
- Other Redeemer
- Other Token Name
- Arbitrary UTXO Datum

- Unbounded protocol value
- Foreign UTXO tokens
- Double or Multiple satisfaction
- Locked Ada
- Locked non Ada values
- Missing UTXO authentication
- UTXO contention

Executive Summary

OpShin is a programming language for developing smart contracts on the Cardano blockchain. It uses valid Python syntax, so developers who know Python can quickly get started. However, OpShin is a simplified, limited version of Python, designed to meet the special needs of blockchain development.

- **Key Components:**
  - ‣ **Type System:** OpShin introduces an `aggressive static type inferencer` to address Python's dynamic typing limitations. Types are inferred, ensuring consistency across variable scopes.
  - ‣ **Compilation Pipeline:** OpShin uses Python's built-in ast library for parsing, eliminating the need for tokenization and AST building. The compilation process involves distinct AST transformations, and the final output is translated into Pluto, an intermediate language, and then compiled into UPLC for on-chain execution.
  - ‣ **Tooling and Debugging:** OpShin provides tools for evaluating scripts in Python, compiling to Pluto for debugging, and generating UPLC for on-chain deployment. It integrates with off-chain libraries, enabling contract deployment and interaction.

The audit focused on the Opshin codebase, ensuring its correctness, security, and efficiency. The scope included the Opshin compiler and its ability to enforce strict Python compliance while generating secure and optimized on-chain code. Notably, the **Pluto to UPLC** (Untyped Plutus Core) compilation process was explicitly **out of scope** for this audit.

Compilation Pipeline

Because Opshin syntax is a subset of valid Python syntax, Opshin uses the Python AST parsing function built into the Python `ast` standard library. This completely eliminates the need to implement the first two steps of the compilation pipeline: tokenization and AST building.

Once an entrypoint is parsed into an AST, 27 distinct AST transformations are applied that weed out syntax and type errors, and gradually transform the AST into something that can easily be converted into *Pluthon*. The last transformation step performs the actual conversion to a *Pluthon* AST. The conversion to the on-chain *UPLC* format is handled by the *Pluthon* library and is out of scope of this library.

Each of the following steps is implemented using a recursive top-down visitor pattern, where each visit is responsible for continuing the recursion of child nodes. This is the same approach that is used in the Python internal codebase.

1. Resolves `ImportFrom` statements respecting the `from <pkg> import *` format, mimicking the Python module resolution behavior to get the module file path, then calling the standard `parse()` method, and recursively resolving nested `from <pkg> import *` statements in the imported modules. This step ignores imports of builtin modules. The `from <pkg> import *` AST node is transformed into a list of statements, all sharing the same scope.

2. Throws an error when detecting a `return` statement outside a function definition.

3. (Optional) Subexpressions that can be evaluated to constant Python values are replaced by their `Constant(value)` equivalents.

4. Removes a deprecated Python 3.8 AST node.

5. Replaces augmented assignment by their simple assignment equivalents. Eg. `a += 1` is transformed into `a = a + 1`.

6. Replaces comparison chains by a series of binary operations. Eg. `a < b < c` is transformed into `(a < b) and (b < c)`.

7. Replaces tuple unpacking expressions in assignments and for-loops, by multiple single assignments. Eg. `(a, b) = <tuple-expr>` is transformed into:

```
1  <tmp-var> = <tuple-expr>
```

```
2  a = <tmp-var>[0]
3  b = <tmp-var>[1]
```

8. Detects `from opshin.std.integrity import check_integrity` and

`from opshin.std.integrity import check_integrity as <name>` statements and injects the `check_integrity` macro into the top-level scope.

9. Ensures that all classes inherit `PlutusData` and that `PlutusData` is imported using `from pycardano import Datum as Anything, PlutusData`.

10. Replaces hashlib functions (`sha256`, `sha3_256`, `blake2b`) imported using `from hashlib import <hash-fn> as <aname>` by raw *Pluthon* lambda function definitions.

11. Detects classes with methods, ensures that `Self` is imported using `from typing import Self`, and adds a class reference to the `Self` AST nodes. Also ensures `List`, `Dict` and `Union` are imported using `from typing import Dict, List, Union`.

12. Throws an error if some of the builtin symbols are shadowed.

13. Ensures that classes are decorated with `@dataclass` and that `@dataclass` is imported using `from dataclasses import dataclass`.

14. Injects the *Pluthon* implementations of a subset of Python builtins before the main module body.

15. Explicitly casts anything that should be boolean (eg. if-then-else conditions, comparison bin ops) by injecting `bool()`.

16. Sets the `orig_name` property of Name, FunctionDef and ClassDef AST nodes.

17. Gives each variable a unique name by appending a scope id.

18. Aggressive Type Inference: Visits each AST node to determine its type, setting its `.typ` property in the process.

19. Turns empty lists into raw *Pluthon* expressions.

20. Turns empty dicts into raw *Pluthon* expressions.

21. Ensures that a function that is decorated with a single named decorator, is decorated with the `@wraps_builtin` decorator, which must be imported using `from opshin.bridge import wraps_builtin`. Such decorated functions are then converted into raw *Pluthon* expressions.

22. Injects the `bytes()`, `int()`, `bool()` and `str()` cast builtins.

23. Removes assignments of types, classes and polymorphic functions (eg. `MyList = List[int]`).

24. (Optional) Iteratively collects all used variables and removes the unused variables. The iteration is stopped if the set of remaining used variables remains unchanged.

25. (Optional) Removes constant statements.

26. Removes Pass AST nodes.

27. Generates the *Pluthon* AST.

Code base

Repository

https://github.com/OpShin/opshin

Commit

d657a227f02670e6b6eed9cac77c0f8a25d51423

Category Classification

- **[S]-Security**: Security focuses on risks that undermine the correctness or safety of programs compiled with the language, where flaws in type systems, serialization logic, or runtime errors could lead to fund loss, lockups, or invalid transactions.

- **[P]-Performance**: Performance examines inefficiencies that inflate transaction fees or breach execution limits, such as unbounded loops or O(n²) algorithms in generated code, excessive Force/Delay wrapping in UPLC output, or missed compile-time optimizations.

- **[M]-Maintainability**: Maintainability evaluates the language implementation's long-term viability, highlighting fragile AST transformations, dead code, or duplicated logic that complicate debugging or feature additions.

- **[U]-Usability**: Usability targets user experience gaps, from cryptic error messages to inconsistent behaviors in compiled output.

Severity Classification

- **Critical**: This vulnerability has the potential to result in significant financial losses to the protocol. They often enable attackers to directly steal assets from contracts or users, or permanently lock funds within the contract.

- **Major**: Can lead to damage to the user or protocol, although the impact may be restricted to specific functionalities or temporal control. Attackers exploiting major vulnerabilities may cause harm or disrupt certain aspects of the protocol.

- **Medium**: May not directly result in financial losses, but they can temporarily impair the protocol's functionality. Examples include susceptibility to front-running attacks, which can undermine the integrity of transactions.

- **Minor**: Minor vulnerabilities do not typically result in financial losses or significant harm to users or the protocol. The attack vector may be inconsequential or the attacker's incentive to exploit it may be minimal.

- **Informational**: These findings do not pose immediate financial risks. These may include protocol optimizations, code style recommendations, alignment with naming conventions, overall contract design suggestions, and documentation discrepancies between the code and protocol specifications.

Finding severity ratings

The following table defines levels of severity and score range that are used throughout the document to assess vulnerability and risk impact

	Level	Severity	Findings
	5	Critical	9
	4	Major	16
	3	Medium	0
	2	Minor	39
	1	Informational	24

Findings

Findings by Security

ID-S501 `List` and `Dict` Comprehension Filters Skip Boolean Type Checks

	Level	**Category**	**Severity**	**Status**
⬛	5	Security	Critical	Pending

Description

The list comprehension type checks in `AggressiveTypeInferencer.list_comprehension()` does not check that the comprehension ifs filter expressions are of boolean type.

If the user inadvertently uses a comprehension filter expression that doesn't evaluate to a bool, a runtime error will always occur if the comprehension generator returns a non-empty list. This can lead to bricked user funds if a validator hasn't been sufficiently tested.

As an example, the following validator will compile without errors, but will always throw a runtime error when the argument is a non-empty list:

```python
1  def validator(a: List[int]) -> None:
2      b = [x for x in a if x]
3      pass
```

Recommendation

We recommend to wrap list comprehension ifs with `Bool` casts in `rewrite_cast_condition.py`.

Resolution

Pending

ID-S502 Type Assertion Wrappers Not Applied on the RHS of BoolOp

Level	Category	Severity	Status
5	Security	Critical	Pending

Description

In `AggressiveTypeInferencer.visit_BoolOp()`, type assertions performed on the left-hand-side don't result in Pluthon AST nodes that convert *UPLC* data types to primitive types.

This leads to unexpected runtime type errors and can potentially result in locked funds at smart contract if the compiled validator isn't sufficiently unit-tested.

The following validator is an example of valid OpShin that will produce *UPLC* that will always fail if the left-hand-side of the and expression is true:

```python
1  from opshin.prelude import *
2
3  def validator(a: Union[int, bytes]) -> None:
4      assert isinstance(a, int) and a == 10
```

Recommendation

Reuse logic related to `self.wrapped` from `AggressiveTypeInferencer.visit_If()`.

Resolution

Pending

ID-S503 Type Assertion Wrappers Not Applied in `while` Statement Bodies

	Level	**Category**	**Severity**	**Status**
■	5	Security	Critical	Pending

Description

In `AggressiveTypeInferencer.visit_While()`, type assertions performed in the `while` statement condition do not result in the addition of Pluthon AST nodes that convert *UPLC* data types to primitive types.

This leads to unexpected runtime type errors, and can potentially lead to smart contract dead-locks if the compiled validator isn't sufficiently unit-tested.

The following validator is an example of valid OpShin that will produce *UPLC* that will always fail if the `while` body is entered:

```python
1  from opshin.prelude import *
2
3  def validator(a: Union[int, bytes]) -> None:
4      while (isinstance(a, int)):
5          if (a > 0):
6              a -= 1
```

Recommendation

Reuse logic related to `self.wrapped` from `AggressiveTypeInferencer.visit_If()`.

Resolution

Pending

ID-S504 `UnionType` Not Implicitly Converted

Level	Category	Severity	Status
5	Security	Critical	Pending

Description

In `PlutoCompiler.visit_Return()` in `compiler.py`, implicit conversion from primitive value to data value is done if the return type is `Any` (i.e. `PlutusData`). This implicit conversion is however not performed when the return type is `Union`.

The type checked AST assumes that functions returning `Union`, always return something correctly converted into `PlutusData`. But that isn't currently being done, leading to a critical bug where the following validator compiles without errors but will always fail during evaluation:

```python
1   from opshin.prelude import *
2
3   def convert(a: int) -> Union[int, bytes]:
4       return a
5
6   def validator(a: Union[int, bytes]) -> Union[int, bytes]:
7       if isinstance(a, int):
8           # In the following the typechecking assumes the return type is `Union[int, bytes]`,
9           # but on-chain it will still be `int` due to missing conversion
10          b = convert(a)
11          if isinstance(b, int):
12              print(str(b))
13
14      return a
```

Similarly, these implicit conversions of `Union` values is missing in `PlutoCompiler.visit_AnnAssign()`.

Recommendation

In `compiler.py`, refactor the `isinstance(typ, AnyType) or isinstance(typ, UnionType)` logic used in `PlutoCompiler.visit_Call()`, and reuse it to check for implicit conversion to data in `PlutoCompiler.visit_Return()` and `PlutoCompiler.visit_AnnAssign()`.

Resolution

Pending

ID-S505 Incorrect Data Conversion to items in `ListType.copy_only_attributes()`

	Level	Category	Severity	Status
■	5	Security	Critical	Pending

Description

In `ListType.copy_only_attributes()` in `type_impls.py`, items are converted to data before being copied, and then converted back to a regular value after being copied. This is wrong, as demonstrated by the following example validator, that compiles successfully, but throws an error when evaluated:

```python
1   from opshin.prelude import *
2   from opshin.std.integrity import check_integrity
3
4   @dataclass
5   class A(PlutusData):
6       d: List[List[int]]
7
8   def validator(d: int) -> None:
9       a: A = A([[d]])
10      check_integrity(a)
11      pass
```

Similarly, this compiles successfully for Dicts nested in Lists, but throws an error when evaluated.

Recommendation

We recommend to remove the conversion to/from data in `ListType.copy_only_attributes()` (i.e. the `transform_ext_params_map(self.typ)(...)` and `transform_output_map(self.typ)(...)` calls).

The `copy_only_attributes()` method of each type should be responsible for its own conversion to/from data. This means the `AtomicType`s (`IntegerType`, `BoolType` etc.) should implement `copy_only_attributes()` to perform the relevant checks, instead of returning the identity function.

This way the `copy_only_attributes()` implementations of `ListType`, `DictType` and `RecordType` don't have to perform explicit conversions of their content, improving maintainability of the codebase.

Resolution

Pending

ID-S506 Missing Length Check in `zip()` Usage

	Level	Category	Severity	Status
■	5	Security	Critical	Pending

Description

In `TupleType.__ge__` in `type_impls.py`, the Python builtin `zip` function is used without check-ing that the lengths of its arguments are the same. This means a shorter length tuple can potentially be passed into a function whose argument expects a longer length tuple.

Though tuples don't yet have a type syntax (thus user-defined functions can't be created that take tuple arguments) tuples can still be used in other ways that lead to compilation succeeding but run-time failures, for example:

```python
1  def validator(a: int) -> int:
2      t1 = (a, a, a)
3      t2 = (a, a)
4
5      t3 = t1 if False else t2
6
7      return t3[2]
```

This example validator will compile successfully but will always fail to run.

Recommendation

Ensure the lengths of the `TupleType`s are the same when comparing them in `TupleType.__ge__`.

Resolution

Pending

ID-S507 Incorrect implementation of `index` method of `ListType`

Level	Category	Severity	Status
5	Security	Critical	Pending

Description

The `index` method, defined in `ListType.attribute()` in `type_impls.py`, uses the wrong built-in method to check item equality. It is currently implemented as `EqualsInteger(x, HeadList(xs))`.

The following example validator compiles successfully, but will always fail to run:

```python
1  from opshin.prelude import *
2
3  def validator(a: Anything, b: Anything) -> int:
4      l: List[Anything] = [a, b]
5
6      return l.index(b)
```

Recommendation

We recommend to change the check to

```
EqualsData(transform_output_map(itemType)(x), transform_output_map(itemType)
(HeadList(xs)))
```
.

Resolution

Pending

ID-S508 `CONSTR_ID` attribute is defined for `Anything` and `Union` of primitives

	Level	**Category**	**Severity**	**Status**
▇	5	Security	Critical	Pending

Description

The following is valid OpShin, but is conceptually strange as it isn't consistent with how attributes are exposed of regular `Union`s (they must exist on each subtype), and can lead to unexpected runtime errors: Both these validators compile successfully, but will always fail to run.

```python
1  from opshin.prelude import *
2
3  def validator(l: List[Anything]) -> int:
4      return l[0].CONSTR_ID
```

```python
1  from opshin.prelude import *
2
3  def validator(u: Union[int, bytes]) -> int:
4      return u.CONSTR_ID
```

Recommendation

- The `CONSTR_ID` attribute for `Anything` can be removed.
- Avoid exposing the `CONSTR_ID` attribute of `Union`s which contain some `non-ConstrData` types.

Resolution

Pending

ID-S509 `FalseData` and `TrueData` uses the wrong `CONSTR_ID`

	Level	**Category**	**Severity**	**Status**
■	5	Security	Critical	Pending

Description

In `ledger/api_v2.py`, `FalseData` uses `CONSTR_ID=1`, and `TrueData` uses `CONSTR_ID=0`.

But according to line 24 of https://github.com/IntersectMBO/plutus/blob/master/plutus-tx/src/PlutusTx/IsData/Instances.hs:

```haskell
1  $(makeIsDataSchemaIndexed ''Bool [('False, 0), ('True, 1)])
```

This mismatch changes the expected behavior of the functions operating on time ranges

Recommendation

Change the `CONSTR_ID` of `FalseData` to 0, and change the `CONSTR_ID` of `TrueData` to 1.

Resolution

Pending

ID-S401 Lack of Namespaced Imports

Level	Category	Severity	Status
🟥 4	Security	Major	Pending

Description

User defined symbols can only be imported using `from <pkg> import *`, and every time such a statement is encountered the complete list of imported module statements is inlined. This can lead to a lot of duplicate statements, and quickly pollutes the global namespace with every symbol defined in every (imported) package.

The following two scenarios explain why this is a critical problem.

- Scenario 1

Imagine both a singular name (eg. `asset`) and a plural name (eg. `assets`) are defined somewhere in the OpShin smart contract codebase or external libraries. The programmer makes a typo and unknowingly uses the wrong variable (e.g. `asset` instead of `assets`). Due to type inference the value of the wrongly used variable might actually have a type that passes the type check (eg. both `asset` and `assets` allow calling `len()`). The program compiles and seems to work even though it doesn't match the programmer's intent.

- Scenario 2

The codebase defines a variable with the same name and type multiple times, but each time another value is assigned. For the programmer it is ambiguous which value will actually be used when referencing the variable. The programmer doesn't know enough about the library code being imported to intuitively figure out which variable shadows all the others.

- Scenario 3

```python
@dataclass()
class Address(PlutusData):
    street: bytes
    city: bytes
    zip_code: int

@dataclass()
class Employee(PlutusData):
    name: bytes
    age: int
    address: Address
```

This code defines a custom class named `Address`, which shadows the built-in Address type from the Cardano ecosystem. It throws a type inference error. However, it should instead show a warning indicating that the name is shadowed.

Recommendation

The current OpShin import mechanism is generally poorly implemented, also for builtins:

- The `hashlib` functions are handled differently from `opshin.std`, yet there is no obvious reason why they should be treated differently.
- The `check_integrity` macro is added to the global scope with its alias name, meaning it suddenly pollutes the namespace of upstream packages.
- Some of the builtin imports suffer from the same issue as imports of user defined symbols: duplication.
- `Dict, List, Union` must be imported in that order from `typing`.
- The `Datum as Anything` import from `pycardano` seems to only exist to help define `Anything` for eg. IDEs, but `Anything` is actually defined elsewhere.

Though the import of builtins will be hidden behind `opshin.prelude` for most users, it is still not implemented in a maintainable way.

A complete overhaul of the import mechanism is recommended, including the implementation of the `import <pkg>` syntax. The OpShin AST should be able to have multiple Module nodes, each with their own scope.

Nice to have:

- Use `.pyi` files for builtin packages, and define the actual builtin package implementation in code in importable scopes.
- OpShin specific builtins should be importable in any pythonic way, even with aliases. Name resolution should be able to figure out the original builtin symbol id/name.
- Detect which python builtins and OpShin builtins are being used, and only inject those.
- Don't expose `@wraps_builtin` decorator.
- Builtin scope entries can be given a "forbid override" flag, instead of having to maintain a list of forbidden overrides in `rewrite/rewrite_forbidden_overwrites.py`.
- Implement a warning for shadowing (instead of e.g. the type inference error thrown in scenario 3). This would help developers catch potential issues early without halting compilation.

An additional advantage of having multiple independent Module AST nodes is that some compilation steps can be multi-threaded.

Resolution

Pending

ID-S201 Custom Function Declarations are Overridden

Level	Category	Severity	Status
🟨 2	Security	Minor	Pending

Description

The code does not validate the source of the `@dataclass` decorator. If a custom dataclass function is defined, it overrides the imported dataclass decorator, and the rewrite transformers does not detect and report this issue.

Example:

```python
1   from dataclasses import dataclass
2
3   # Custom dataclass decorator
4   def dataclass(cls):
5       return cls
6
7   # Refers to the custom decorator, not the one from 'dataclasses'
8   @dataclass
9   class MyClass(PlutusData):
10
11  def validator(a: int) -> None:
12      return None
```

The code checks for the presence of the `@dataclass` decorator and validates that `dataclass` is imported from the package `dataclasses`, but does not verify or report if the decorator is overridden by a custom function.

Recommendation

1. To ensure that function names are also not overridden in addition to variable names, we recommend to extend the `RewriteForbiddenOverwrites` transformer to check for forbidden names in function definitions. This will ensure that function names do not conflict with reserved or forbidden names.

2. Raise a descriptive warning if any custom definitions are detected, e.g., In this case "The dataclass function can't override the existing import".

Resolution

Pending

Findings by Performance

ID-P401 Redundant Bound External Variables Passing in Function Calls

	Level	Category	Severity	Status
■	4	Performance	Major	Pending

Description

In `PlutoCompiler.visit_Call()` in `compiler.py`, `bound_vs` includes all external variables referenced inside a function, which are then passed as the initial arguments of the function whenever it is called. This is unnecessary and can become extremely expensive.

In the following example, `add` is an external variable that is being referenced inside `validator`:

```python
1  def add(a: int, b: int) -> int:
2      return a + b
3
4  def validator(a: int, b: int) -> int:
5      return add(a, b)
```

Compiling this validator with `opshin compile_pluto validator.py -O3`, produces:

```pluto
1  (\
2    1val_param0 1val_param1 -> (
3      let
4        a_1 = (# (Error ((! Trace) 'NameError: a' ()))));
5        a_2 = (# (Error ((! Trace) 'NameError: a' ()))));
6        add_0 = (# (Error ((! Trace) 'NameError: add' ()))));
7        b_1 = (# (Error ((! Trace) 'NameError: b' ()))));
8        b_2 = (# (Error ((! Trace) 'NameError: b' ()))));
9        validator_0 = (# (Error ((! Trace) 'NameError: validator' ()))))
10     in (
11       let add_0 = (# (
12         \a_1 b_1 -> (
13           (\
14             1self 1other -> (AddInteger 1self 1other)
15           ) (! a_1) (! b_1)
16         )
17       )) in (
```

```
18        let validator_0 = (# (\
19          add_0 a_2 b_2 -> (
20            let
21              1p0 = (! a_2);
22              1p1 = (! b_2)
23            in (
24              (! add_0) (# 1p0) (# 1p1)
25            )
26          )
27        )) in (
28          IData (
29            let
30              1p0 = (UnIData 1val_param0);
31              1p1 = (UnIData 1val_param1)
32            in (
33              (! validator_0) add_0 (# 1p0) (# 1p1)
34            )
35          )
36        )
37      )
38    )
39  )
40 )
```

Note the redundant passing around of `add_0` as the first argument of `validator_0`.

Recommendation

OpShin does not seem to support mutual recursion, so it might not even be necessary to pass all bound vars as arguments to the functions if the functions simply maintain their order in the final *UPLC.*

Alternatively, if the order of the functions changes in the final *UPLC,* filter out the bound vars that are naturally available as part of the outer scope of the function.

Resolution

Pending

ID-P402 Excessive `Force`/`Delay` Use for User-Defined Variables

	Level	Category	Severity	Status
🟥	4	Performance	Major	Pending

Description

Notably in `PlutoCompiler.visit_ClassDef()` in `compiler.py`, the class constructor function is wrapped in a `Delay` term. This is unnecessary as it a simple `Lambda` term, and doesn't throw an error nor incur a cost when evaluated by the *UPLC* CEK machine.

The architecture of the OpShin compiler currently requires every user-defined variable to be wrapped with `Delay`. Upon referencing those variables, a `Force` term is added. This leads to a small amount of overhead almost everywhere in the OpShin generated *UPLC*.

Recommendation

We recommend updating the compiler to automatically detect when user-defined variables refer to Lambda functions using the type information available in the OpShin AST, so that explicit `Delay`/`Force` wrapping is no longer required for UPLC variables.

Resolution

Pending

ID-P403 `NameError` Expressions Added to Each Loaded Variable

	Level	**Category**	**Severity**	**Status**
	4	Performance	Major	Pending

Description

During the code generation step, in `PlutoCompiler.visit_Module()` in `compiler.py`, a `NameError` expression is added for each loaded variable. This set of variables potentially includes each and every variable defined in the program, and thus significantly bloats the generated code. The optimizations built into OpShin don't seem to be able to eliminate this bloat.

The benefit of these `NameError` expressions is that runtime debugging is easier if a variable is referenced that doesn't actually exist. However, the compiler should be able to detect such situations at compile time and avoid generating dead code.

The OpShin *Pluthon → UPLC* compilation step isn't able to eliminate these `NameError` expressions, even at optimization level 3.

Recommendation

Add a compiler flag so that these `NameError` expressions aren't added to the generated *UPLC* code.

Resolution

Pending

ID-P201 Redundant Explicit Cast to Boolean

Level	Category	Severity	Status
    2	Performance	Minor	Pending

Description

The `RewriteConditions` transformer explicitly rewrites all conditions (e.g., in if, while, assert, etc.) to include an implicit cast to bool using a special variable, `SPECIAL_BOOL`. However, this is redundant in cases such as:

1. The condition is already a boolean (e.g., if True or if x == y where the result is already a boolean).
2. The condition is a constant node (e.g., if True or if False).

In such cases, adding an explicit cast to bool is unnecessary and can degrade performance, especially in cases where the condition is evaluated repeatedly (e.g., in loops).

Recommendation

Modify the `RewriteConditions` transformer in `rewrite/rewrite_cast_condition.py` to skip the explicit cast to bool when the condition is already a boolean and a constant node.

Resolution

Pending

ID-P202 Irrelevant *UPLC* builtins in Output

Level	Category	Severity	Status
2	Performance	Minor	Pending

Description

```python
1  def validator(datum: bytes, redeemer: None, context: ScriptContext) -> None:
2      assert datum[0] == 0, "Datum must start with null byte"
```

Compiling this OpShin code using both the default optimiser and the aggressive optimiser (-O3 optimization flag) resulted in the same output. It includes built-in functions such as `addInteger`, `lessThanInteger`, and `lengthOfByteString`, which seem irrelevant for checking equality of the first byte of the datum (`ByteString`) with `0`.

Recommendation

Inspect if this coming from the *Pluthon* conversion and try to avoid unnecessary builtins as part of optimization.

Resolution

Pending

ID-P203 Key Data Value Conversion is Loop Invariant

	Level	Category	Severity	Status
🟨	2	Performance	Minor	Pending

Description

In `PlutoCompiler.visit_Subscript()` in `compiler.py`, in the Pluthon code generation of the dict key indexing, `transform_output_map(dict_typ.key_typ)(OVar("key"))` doesn't change during the search loop.

Recommendation

We recommend assigning `transform_output_map(dict_typ.key_typ)(OVar("key"))` to a temporary variable outside the loop to avoid redundant computations and improve efficiency.

Resolution

Pending

ID-P204 Double Iteration in `hex` and `oct` Methods

	Level	Category	Severity	Status
🟨	2	Performance	Minor	Pending

Description

In `type_impls.py`, the `hex` method of `ByteStringType` performs two loops. The first loop converts the bytestring to a list of integers, and the second loop converts this list to a list of ASCII characters.

Similarly in `fun_impls.py`, the `hex` and `oct` functions perform two loops.

UPLC loops have non-negligible overhead, and merging these two loops into a single loop will give some performance benefit.

Recommendation

Merge the two loops of the `hex` method of `ByteString`, and the `hex` and `oct` functions in `fun_impls.py`, into one loop.

Resolution

Pending

ID-P205 Double Loop in `int` Method for String Parsing

	Level	Category	Severity	Status
   	2	Performance	Minor	Pending

Description

In `type_impls.py`, the `IntImpl` class generates *UPLC* code that performs two loops. The first loop creates a range sequence, and the second loop uses the range from the first loop to iterate over the string being parsed.

Recommendation

Due to *UPLC* loop overhead, merging these two loops into a single loop will provide a performance benefit.

Resolution

Pending

ID-P206 No Short-Circuiting in `all` and `any` Builtins

	Level	Category	Severity	Status
🟨	2	Performance	Minor	Pending

Description

In `fun_impls.py` , the `all` builtin keeps iterating to the end of the boolean list, even if a `false` value has already been encountered. Similarly, the `any` builtin keeps iterating even if a `true` value has already been encountered.

Recommendation

Use a variant of the Pluthon `FoldList` function to exit the iteration prematurely when `all` or `any` encounter a `False` or `True` value respectively.

Resolution

Pending

ID-P207 Unnecessary Double Conversion in Annotated Data Assignments

	Level	Category	Severity	Status
🟨	2	Performance	Minor	Pending

Description

In `PlutoCompiler.visit_AnnAssign()` in `compiler.py`, data values on the right-hand-side are implicitly converted to primitive values. Subsequently primitive values are implicitly converted to data values depending on the left-hand-side type annotation.

This potentially leads to a double conversion (data -> primitive -> data) if the left-hand-side type annotation is a data type.

The double conversion does not introduce significant overhead, as it results in two wrapped identity functions during code generation; however, it is still unnecessary.

Recommendation

We recommend skipping implicit conversions when both the right-hand side and the left-hand side are already data values.

Resolution

Pending

ID-P208 `POWS` List Always Accessed in Reverse Order

	Level	Category	Severity	Status
🟨	2	Performance	Minor	Pending

Description

In `std/bitmap.py`, the `POWS` list is always accessed in reverse order:

```python
1 POWS[(BYTE_SIZE - 1) - (i % BYTE_SIZE)]
```

The `POWS` list can be reversed instead, allowing the elimination of the `(BYTE_SIZE - 1) -` operation.

Recommendation

Reverse `POWS` during its assignment using the `reversed()` builtin, then remove the `(BYTE_SIZE - 1) -` operation wherever `POWS` is accessed.

Resolution

Pending

ID-P101 Optimization Not Showing the Result of Execution

	Level	Category	Severity	Status
	1	Performance	Informational	Pending

Description

As there is no equivalent for the `check_integrity` function in Python, the optimizer isn't able to perform it and and instead just outputs the result of compilation.

```python
1  @dataclass()
2  class B(PlutusData):
3      CONSTR_ID = 1
4      foobar: int
5      bar: int
6
7  def validator(x: B) -> None:
8      x = B(4,5)
9      check_integrity(x)
```

For this code, the *UPLC* outputs the compiled code for both branches of the builtin function `ifThenElse`.

```
1  (lam 1x [(lam 1x (force [[[(force (builtin ifThenElse)) [[(builtin
   equalsData) 1x] [0p_AdHocPattern_6e5e35746e0db09c0956f182750126a838d5650add53b85{
   1x]]] (delay (con unit ()))] (delay [(lam _ (error)) [[(force (builtin
   trace)) (con string "ValueError: datum integrity check failed")]]])])))])
```

.

Recommendation

If feasible, explore the possibility of implementing a Python-side equivalent of the `check_integrity` function to align Python evaluation behavior with on-chain execution.

Resolution

Pending

Findings by Maintainability

ID-M401 No Copies of Middle Expression

	Level	Category	Severity	Status
🟥	4	Maintainability	Major	Pending

Description

When rewriting chained comparisons to individual comparisons combined with `and` for e.g.

```python
1   <expr-a> < <expr-b> < <expr-c> to (<expr-a> < <expr-b>) and (<expr-b> <
    <expr-c>)
```

in `rewrite/rewrite_comparison_chaining.py`, no copies of `<expr-b>` seem to be created, leading to the same AST node instance appearing twice in the AST.

The compiler steps frequently mutate the AST nodes instead of creating copies, which can make debugging issues more difficult in this case.

Recommendation

Similar to `rewrite/rewrite_tuple_assign.py`, create temporary variables for each of the middle expressions in the chain. Then refer to those temporary variables in the resulting BinOp expressions.

This approach avoids the issue described and also avoids the recalculation of the same expression (potentially expensive).

Resolution

Pending

ID-M402 Compiler Step 22 Doesn't do Anything

	Level	Category	Severity	Status
🟥	4	Maintainability	Major	Pending

Description

Step 22 of the compilation pipeline (as detailed on page 9) is designed to inject builtins like `bool()`, `bytes()`, `int()`, and `str()` as `RawPlutExprs`. However, the internal types (i.e., .constr_type()) of these builtins are inherently polymorphic (`PolymorphicFunctionType`), causing them to be skipped immediately. This check is either redundant or may be intended for a future use case that hasn't been implemented yet. Currently, this step adds no value to the compilation process.

Recommendation

Get rid of compiler step 22, thus getting rid of `rewrite/rewrite_inject_builtin_constr.py`.

Resolution

Pending

ID-M403 Alias Names for Imports

Level	Category	Severity	Status
4	Maintainability	Major	Pending

Description

In both `rewrite_import_hashlib.py` and `rewrite_import_integrity_check.py`, there is a potential issue with name conflicts when handling aliased imports.

1. rewrite_import_hashlib.py:

The transformer handles aliased imports but does not explicitly check for name conflicts with existing variables or functions in the scope.

Currently, if a conflict occurs, as in the code below, it throws a type inference error. It does not provide a clear or user-friendly error message about the name conflict.

```python
1  from hashlib import sha256 as hsh
2
3      x = hsh(b"123").digest()
4      hsh = b"opshin"
```

2. rewrite_import_integrity_check.py:

When an alias is used (e.g., import check_integrity as ci), the alias name (ci) is added to `INITIAL_SCOPE` as a new key-value pair.

There is no explicit check to ensure that the alias does not conflict with existing names in `INITIAL_SCOPE`.

This could lead to unintended overwriting of existing variables, causing subtle bugs or unexpected behavior.

Recommendation

To address these issues, the following improvements are recommended:

- Before adding an aliased import to the scope, explicitly check if the alias (or the original name, if no alias is provided) already exists in the scope.

- If a conflict is detected, raise a clear and descriptive error indicating the name conflict and suggesting a resolution (e.g., using a different alias).

Resolution

Pending

ID-M404 Unable to Loop Over Tuple

	Level	Category	Severity	Status
	4	Maintainability	Major	Pending

Description

According to `AggressiveTypeInferencer.visit_For()`, the following validator should be valid:

```python
1  def validator(_: None) -> None:
2      t = (1, 2)
3      for x in t:
4          pass
```

Instead the compiler throws the following non user-friendly error:

`'InstanceType' object has no attribute 'typs'`.

Recommendation

The PlutoCompiler doesn't actually allow iterating over tuples using for loops.

Either remove the tuple related type checks in `AggressiveTypeInferencer.visit_For()` and throw a more explicit error, or implement the necessary code generation that allows iterating over tuples in `PlutoCompiler.visit_For()`.

Resolution

Pending

ID-M405 Fragile `scopes` and `wrapped` Handling in AggressiveTypeInferencer

	Level	**Category**	**Severity**	**Status**
🟥	4	Maintainability	Major	Pending

Description

The way `self.scopes` and `self.wrapped` are mutated/restored inside AggressiveTypeInferencer gives fragile and duplicate code.

Recommendation

Pass a context object as a separate argument through all the `visit_<Node-type>()` methods. The context object contains the current scope and type assertion information like `wrapped`, and links to parent scopes.

Resolution

Pending

ID-M406 Unnecessary Data Construction for Void Validators

Level	Category	Severity	Status
4	Maintainability	Major	Pending

Description

```python
1 def validator(x:int):
2     assert x == 1
```

For a simple validator with no return value, as shown above, the *UPLC* constructs data for integer 0 in addition to nil data, which isn't necessary and does not go away even after optimization.

```
[(lam v0 [(lam v1 [(lam v2 (lam v3 [(lam v4 [(lam v5 [(lam v6 [(lam v7 [(lam v8 [[(force
v7) v6] (delay v8)]) [(builtin unIData) v3]]) (delay (lam v9 (lam v10 (force [[[(force
(builtin ifThenElse)) [(lam v11 [v1 (delay v11)]) [[v2 (force v10)] (con integer
1)]]] (delay [[(builtin constrData) (con integer 0)] [(builtin mkNilData) (con unit
())]])] (delay [(lam v12 (error)) (con unit ())])])))]) (delay [(lam v13 (error))
[[(force (builtin trace)) (con string "NameError: ~bool")] (con unit ())]])]) (delay
[(lam v14 (error)) [[(force (builtin trace)) (con string "NameError: x")] (con unit
())]])]) (delay [(lam v15 (error)) [[(force (builtin trace)) (con string "NameError:
validator")] (con unit ())]])]))) (builtin equalsInteger)]) [(lam v0 (lam v16 [(lam
v17 v17) (force v16)])) (builtin equalsInteger)]]) (builtin equalsInteger)]
```
.

Recommendation

Optimize the codebase to avoid data construction for void validators.

Resolution

Pending

ID-M201 Compiler Version Inconsistency

	Level	Category	Severity	Status
🟨	2	Maintainability	Minor	Pending

Description

The compiler version is defined explicitly in both `pyproject.toml` and `opshin/__init__.py`, which can lead to accidental mismatch if the maintainers of OpShin forget to update either.

Recommendation

According to <u>StackOverflow</u>, the following change to `__init__.py` might be enough:

```python
1  import importlib.metadata
2  __version__ = importlib.metadata.version("opshin")
```

Resolution

Pending

ID-M202 Implicit Import of `plt` in `compiler.py`

	Level	Category	Severity	Status
🟨	2	Maintainability	Minor	Pending

Description

In `compiler.py` :

- `plt` is made available by importing all from `type_inference`
- and inside `type_inference.py` importing all from `typed_ast`
- and inside `typed_ast.py` importing all from `type_impls`
- and finally inside `type_impls.py` importing all from `util` .

At the same time `CompilingNodeTransformer` and `NoOp` are imported directly from `util` .

Recommendation

Consistently use named imports in whole compiler codebase.

Resolution

Pending

ID-M203 TypedModule Dependency Before Type Inference

	Level	Category	Severity	Status
	2	Maintainability	Minor	Pending

Description

The `RewriteInjectBuiltins` transformer operates on `TypedModule` nodes, which are expected to be available only after aggressive type inference has occurred. However, this transformer is part of the compilation process that runs before type inference is complete. This creates a logical inconsistency, as `TypedModule` nodes are not guaranteed to exist at this stage.

Recommendation

Refactor the transformer to work with untyped or partially typed nodes until type inference is complete. Alternatively, ensure that this step is moved to a later stage in the compilation process, where `TypedModule` nodes are guaranteed to exist.

Resolution

Pending

ID-M204 Inconsistent Handling of Polymorphic Functions

	Level	Category	Severity	Status
	2	Maintainability	Minor	Pending

Description

The code uses two different approaches to identify and skip polymorphic functions:

Case 1: Checks if `b.value` is not an instance of `plt.AST`:

```python
1  if not isinstance(b.value, plt.AST):
2      continue
```

Case 2: Checks if the type of the function is `PolymorphicFunctionType`:

```python
1  if isinstance(typ.typ, PolymorphicFunctionType):
2      continue
```

This dual approach makes the code harder to understand. Additionally, polymorphic functions can only be definitively identified after type checking, which further complicates the logic.

Recommendation

1. Unify the logic for identifying polymorphic functions.

2. Since polymorphic functions can only be definitively identified after type checking, consider moving the logic of `rewrite/rewrite_inject_builtins.py` to a later stage in the compilation process, where type information is fully available.

Resolution

Pending

ID-M205 Relative Imports Not Supported

	Level	Category	Severity	Status
▮	2	Maintainability	Minor	Pending

Description

Relative imports (e.g., from .module import x) are not supported because the package parameter is always set to `None` in the method `import_module` in `rewrite/rewrite_import.py`. This was tested by creating two files inside a package like below and they did not work.

```python
# example_module.py
from opshin.prelude import *

@dataclass
class ExampleClass(PlutusData):
    CONSTR_ID = 0
    pubkeyhash: PubKeyHash

def validator():
    pass
```

```python
# example_relativeimport.py
from .example_module import ExampleClass

def validator():
    obj = ExampleClass(pubkeyhash = "12344")
    print("Rewrite import test:", obj)
```

Recommendation

1. Modify the code to handle relative imports by correctly setting the package parameter according to the code.

2. Add documentation clarifying how to use relative imports.

Resolution

Pending

ID-M206 Missing Support for Annotated Variable Nodes in `rewrite_orig_name.py`

	Level	**Category**	**Severity**	**Status**
🟨	2	Maintainability	Minor	Pending

Description

The logic in `rewrite/rewrite_orig_name.py` currently checks for `Name`, `ClassDef`, and `FunctionDef` nodes but does not account for annotated variable assignments (e.g., x: int = 10). These nodes (`AnnAssign` in AST terms) may also contain a pointer to the original name.

Recommendation

Extend the node-checking logic to include `AnnAssign`.

Resolution

Pending

ID-M207 Similar Logic in `visit_ListComp()` and `visit_DictComp()`

	Level	**Category**	**Severity**	**Status**
🟨	2	Maintainability	Minor	Pending

Description

In `PlutoCompiler` in `compiler.py`, the `visit_ListComp()` and `visit_DictComp()` methods are very similar.

Recommendation

Refactor and reuse the common functionality of `PlutoCompiler.visit_ListComp()` and `PlutoCompiler.visit_DictComp()`.

Resolution

Pending

ID-M208 Incorrect Type Annotation in `Type.binop` and `Type._binop_bin_fun`

	Level	**Category**	**Severity**	**Status**
🟨	2	Maintainability	Minor	Pending

Description

The type annotations of the `Type.binop` and `Type._binop_bin_fun` methods in `type_impls.py` contains a mistake: `AST` should be `TypedAST`.

Recommendation

Change the type annotation of the `other` argument in `Type.binop` and `Type._binop_bin_fun` from `AST` to `TypedAST`.

Resolution

Pending

ID-M209 Similar Logic in `cmp()` Methods of `RecordType` and `UnionType`

	Level	Category	Severity	Status
   	2	Maintainability	Minor	Pending

Description

In `type_impls.py`, the implementations of `RecordType.cmp()` and `UnionType.cmp()` are almost exact copies of `AnyType.cmp()`.

Recommendation

Refactor and reuse the logic of `AnyType.cmp()` for `RecordType.cmp()` and `UnionType.cmp()`.

Resolution

Pending

ID-M210 `super.binop_bin_fun()` Not Called

	Level	Category	Severity	Status
   	2	Maintainability	Minor	Pending

Description

In `type_impls.py`, the `_binop_bin_fun()` method implementations don't fall through to calling the `_binop_bin_fun()` method of the `Type` ancestor class.

Recommendation

Fall through to calling `super._binop_bin_fun()`, so that the associated `"Not implemented"` error is thrown.

Resolution

Pending

ID-M211 Shared Logic in `oct` and `hex` Builtins

	Level	Category	Severity	Status
🟨	2	Maintainability	Minor	Pending

Description

In `fun_impls.py`, the `oct` builtin uses exactly the same logic as `hex`, except that the base is different (8 vs 16).

Recommendation

Refactor and reuse the code generation logic of `hex` for `oct`.

Resolution

Pending

ID-M212 Unable to Use Negative Index Subscripts

	Level	Category	Severity	Status
	2	Maintainability	Minor	Pending

Description

In `PlutoCompiler.visit_Subscript()` in `compiler.py`, literal negative indices for tuples and pairs aren't detected as being a `Constant` AST node.

Other parts of the codebase do however allow handling negative indices, but using such a literal negative index for tuples and pairs will always throw an error at this (late) compilation stage.

Recommendation

Whenever checking that a subscript is `Constant`, ensure it isn't negative (so that if future versions of the Python tokenizer treat literal negative numbers as `Constant`, this doesn't break OpShin).

Alternatively detect negative indexes correctly in

`AggressiveTypeInferencer.visit_Subscript()` in `type_inference.py`.

Resolution

Pending

ID-M101 Migrate Some Utility Functions

	Level	Category	Severity	Status
	1	Maintainability	Informational	Pending

Description

Some utility functions defined in the OpShin library would make more sense as part of the *UPLC* or *Pluthon* packages.

- `rec_constant_map_data()` and `rec_constant_map()` (defined in `opshin/compiler.py`) can be moved to the *UPLC* package.
- `to_uplc_builtin()` and `to_python()` (defined in `opshin/bridge.py`) can also be moved to the *UPLC* package.
- `OVar()`, `OLambda()`, `OLet()`, `SafeLambda()`, `SafeOLambda()` and `SafeApply()` (defined in `opshin/util.py`) can be moved to the *Pluthon* package.

Recommendation

We recommend reorganizing the codebase by moving utility functions to the packages where they logically belong.

Resolution

Pending

ID-M102 `PlutoCompiler.visit_Pass` is Redundant

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

Compiler step 26 removes the Pass AST node, but step 27 (the *Pluthon* code generation step) defines a `visit_Pass` method that seems to return the identity function.

Recommendation

Remove the `visit_Pass` method. If step 26 fails to remove all Pass AST nodes, then the `PlutoCompiler` will throw a "Can not compile Pass" error, instead of masking the improper implementation of step 26.

Resolution

Pending

ID-M103 Incorrect Return Type in some `TypeCheckVisitor` methods

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

`visit_BoolOp()` and `visit_UnaryOp()` use `PairType` as the return type annotation, but actually return tuples.

Recommendation

Change the return type of `visit_BoolOp()` and `visit_UnaryOp()` from `PairType` to `TypeMapPair`.

Resolution

Pending

ID-M104 Refactor `self.wrapped` Reset in `AggressiveTypeInferencer`

	Level	Category	Severity	Status
	1	Maintainability	Informational	Pending

Description

`visit_IfExp()` and `visit_If()` (and once finding S503 is resolved, `visit_While()`) contain the following (duplicate) lines of Python code:

```python
1 self.wrapped = [x for x in self.wrapped if x not in prevtyps.keys()]
```

Besides being duplicate, the `x not in prevtyps.keys()` expression can be replaced by `x not in prevtyps`.

Recommendation

Extract the logic for resetting `self.wrapped` into a dedicated method in `AggressiveTypeInferencer` to avoid duplication, and replace `prevtyps.keys()` by `prevtyps`.

Resolution

Pending

ID-M105 Redundant Code in `AggressiveTypeInferencer`

	Level	Category	Severity	Status
	1	Maintainability	Informational	Pending

Description

In `AggressiveTypeInferencer.visit_sequence()`, the `arg.annotation is None` test in the second assertion is redundant, as the surrounding `if` statement test already ensures this is always false.

Recommendation

Remove the redundant check in the second assertion in

`AggressiveTypeInferencer.visit_sequence()` in `type_inference.py`.

Resolution

Pending

ID-M106 Spread-Out `not in` Handling in AggressiveTypeInferencer

	Level	**Category**	**Severity**	**Status**
	1	Maintainability	Informational	Pending

Description

In `AggressiveTypeInferencer.dunder_override()`, `not in` is treated as `in`, and `not` is treated as `__bool__`. Then in `visit_Compare()` and `visit_UnaryOp()` respectively this is compensated for by wrapping the AST node returned by the `dunder_override()` method with a `Not` AST node.

So logic that is inherently related to `dunder_override()` is spread over two other functions as well.

Recommendation

Return the final AST node from `dunder_override()`, so the explicit wrapping with a `Not` AST node doesn't become the responsibility of the call site.

Resolution

Pending

ID-M107 Immediate Access of Recently Appended List Item

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

In `AggressiveTypeInferencer.visit_BoolOp()`, child nodes visited and the returned typed AST nodes are appended to a `values` list, the appended value is then immediately referenced as `values[-1]`.

Recommendation

Assign the return typed AST nodes to a variable, and reference that variable in the subsequent line of code where the type checks are generated.

Resolution

Pending

ID-M108 Inconsistent Treatment of Tuple Slicing

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

`AggressiveTypeInferencer.visit_Subscript()` supports tuple slicing, but

`PlutoCompiler.visit_Subscript()` does not.

Recommendation

In the TupleType branch in

`AggressiveTypeInferencer.visit_Subscript()` remove the nested branch with the condition that reads `all(ts.value.typ.typ.typs[0] == t for t in ts.value.typ.typ.typs)`.

Resolution

Pending

ID-M109 `RecordReader.extract()` Doesn't Need to be Static

	Level	Category	Severity	Status
▮	1	Maintainability	Informational	Pending

Description

`RecordReader.extract()`, in `type_inference.py`, is a static method using the `@classmethod` decorator. This leads to unnecessary indirection when this method is called.

Recommendation

Instantiate the `RecordReader` directly with an argument of `AggressiveTypeInferencer` type, and change `extract()` to be a regular method (internally changing `f` to `self`).

Resolution

Pending

ID-M110 Assumption of spec for the Parent Module in `rewrite_import.py`

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

1. The code assumes that `__spec__` is always available for the parent module. However, this may not always be true, especially in dynamically created modules.

2. The code does not handle cases where `spec.loader.exec_module` fails to load the module.

Recommendation

1. Provide a fallback mechanism or raise a more descriptive error message if **spec** is missing, ensuring the code does not fail silently.

2. Wrap the call `spec.loader.exec_module(module)` in a try-catch block and log or raise an appropriate error message to help diagnose issues when module loading fails.

Resolution

Pending

ID-M111 Iterating Over `sys.modules` Safely

	Level	Category	Severity	Status
▆	1	Maintainability	Informational	Pending

Description

The code does not follow the Python documentation's recommendation to use `sys.modules.copy()` or `tuple(sys.modules)` when iterating over `sys.modules`. This can lead to exceptions if the size of `sys.modules` changes during iteration due to code execution or activity in other threads.

Recommendation

Replace any direct iteration over `sys.modules` with `sys.modules.copy()` or `tuple(sys.modules)` to avoid potential runtime exceptions.

Ensure that all iterations over `sys.modules` are thread-safe and do not cause side effects during execution.

Resolution

Pending

ID-M112 Replace Repeated `Unit` Definition with a New Variable

	Level	Category	Severity	Status
	1	Maintainability	Informational	Pending

Description

`plt.ConstrData(plt.Integer(0), plt.EmptyDataList())` is used in several places as the `PlutusData` equivalent of `Unit` (i.e. `None` in Python/Opshin):

- once in `PlutoCompiler.visit_FunctionDef()` in `compiler.py`.
- twice in `PlutoCompiler.visit_Module()` in `compiler.py`.
- once in `TransformOutputMap` in `type_impls.py`.

Recommendation

Assign `plt.ConstrData(plt.Integer(0), plt.EmptyDataList())` to a new variable named `Void` (or another appropriate name), and reuse that instead.

Resolution

Pending

ID-M113 Enforce Non-Null Slice Bounds in Typed AST

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

In `PlutoCompiler.visit_Subscript()` in `compiler.py`, in list slice indexing, the possibility of `lower==None` and `upper==None` isn't taken into account here, even though the Python types of the `TypedSubscript.slice.lower` and `TypedSubscript.slice.upper` fields still allows `None`.

In `type_inference.py`, `TypedSubscript.slice.lower` and `TypedSubscript.slice.upper` are ensured to be defined. The resulting typed AST never contains unset slice ranges.

Recommendation

In `typed_ast.py`,

annotate that `TypedSubscript.slice.lower` and `TypedSubscript.slice.upper` can't be `None`.

Resolution

Pending

ID-M114 Standardize Constructor Index Variable Name

	Level	**Category**	**Severity**	**Status**
	1	Maintainability	Informational	Pending

Description

In `UnionType.stringify()` in `type_impls.py`, `c` is used as a temporary variable for the constructor index, but in other places `constr` is used.

Recommendation

Change `c` to `constr` so that `constr` is used consistently as the name of the Pluthon variable containing the constructor index.

Resolution

Pending

ID-M115 Typo in Assertion Message

Level	Category	Severity	Status
1	Maintainability	Informational	Pending

Description

In `BytesImpl` in

`type_impls.py`, the second assertion reads "Can only create bools from instances."

Recommendation

Change the error message to `"Can only create bytes from instances"`.

Resolution

Pending

ID-M116 Unclear When a Python `Constant` can be `PlutusData`

	Level	**Category**	**Severity**	**Status**
	1	Maintainability	Informational	Pending

Description

In function `rec_constant_map()` in `compiler.py`, the value of the `Constant` AST node can apparently be `PlutusData`. It is unclear where or how this is used. `PlutusData` `Constant` values possibly result from evaluation in `optimize_const_folding.py`, but also this is unclear.

Recommendation

Add a comment to `rec_constant_map()` explaining where `PlutusData` comes from.

Resolution

Pending

Findings by Usability

ID-U401 Type Safe Tuple Unpacking

	Level	Category	Severity	Status
■	4	Usability	Major	Pending

Description

Tuple unpacking (Step 7 in compilation pipeline) is currently being rewritten before the ATI (aggressive type inference) step. This allows writing unpacking assignments with a mismatched number of tuple entries.

If there are more names on the left side, this throws a non-user-friendly `FreeVariableError`. If there are less, the rewritten code is valid, even though in Python it wouldn't be valid, thus violating the expected "strict subset of Python" behavior.

There might be other ways this can be abused to get inconsistent behavior.

Recommendation

Perform this step after type inference. Check tuple types during type inference.

Resolution

Pending

ID-U402 Non-friendly Error Message while Using Wrong Import Syntax

Level	Category	Severity	Status
4	Usability	Major	Pending

Description

Using `import <pkg>` or `import <pkg> as <aname>` isn't supported and throws a non-user-friendly error: "free variable '<pkg-root>' referenced before assignment in enclosing scope."

Recommendation

Improve the error message to say that the syntax is wrong and hinting at the correct syntax.

Resolution

Pending

ID-U403 `bytes.fromhex()` Does Not Work

	Level	**Category**	**Severity**	**Status**
🟥	4	Usability	Major	Pending

Description

The OpShin documentation mentions the existence of the `bytes.fromhex()` static method.

The following snippet doesn't compile though:

```python
def validator(_: None) -> None:
    bs = bytes.fromhex("0123")
    assert len(bs) == 2
```

The compiler throws the following error: `Can only access attributes of instances`.

Recommendation

Either ensure attributes of builtin types like `bytes` can actually be accessed, or remove `bytes.fromhex()` from the OpShin documentation.

Resolution

Pending

ID-U404 Inconsistent Errors for Duplicate `CONSTR_ID` in `Union`s

	Level	Category	Severity	Status
■	4	Usability	Major	Pending

Description

The following example validator compiles without error:

```python
1   from opshin.prelude import *
2
3   @dataclass()
4   class A(PlutusData):
5       CONSTR_ID = 1
6       a: bytes
7
8   @dataclass()
9   class B(PlutusData):
10      CONSTR_ID = 2
11      a: int
12      b: int
13
14  @dataclass()
15  class C(PlutusData):
16      CONSTR_ID = 2
17      a: int
18      b: int
19
20  def validator(_: Union[Union[A, B], C]) -> None:
21      pass
```

Only after if the fields of `C` are changed (e.g. changing the name of field `b` to `c`), does the compiler throw the expected error:

`Union must combine PlutusData classes with unique constructors` .

Changing the annotation in the example to `Union[A, B, C]` (while keeping the fields of `B` and `C` the same) gives the following compiler error:

`Duplicate constr_ids for records in Union: {'A': 1, 'B': 2, 'C': 2}` .

Now consider the following modified validator using the same three classes:

```python
def validator(x: Union[Union[A, B], C]) -> None:
    assert isinstance(x, C)
```

Compiling this example gives the following non user-friendly error:

`Trying to cast an instance of Union type to non-instance of union type`.

Recommendation

Fix these error inconsistencies by detecting duplicate `CONSTR_ID`s after flattening the `Union` in `union_types()` in `type_inference.py`. Detect duplicates based on `CONSTR_ID` alone, and not based on data field equivalence.

Resolution

Pending

ID-U405 Non User-friendly Error while Calling `str()` on a `Union`

	Level	Category	Severity	Status
■	4	Usability	Major	Pending

Description

Consider the following example validator:

```python
1  def validator(a: Union[int, bytes]) -> None:
2      assert str(a) == "0"
```

Compiling this example gives the following error:

`'IntegerType' object has no attribute 'record'`.

Recommendation

Generalize the code generation in `UnionType.stringify()` in `type_impls.py`, so that it works for any combination of `int`, `bytes`, `List[Anything]` or `Dict[Anything, Anything]`.

Resolution

Pending

ID-U406 Inconsistent Type Inference of Literal `lists` and `dicts`

Level	Category	Severity	Status
4	Usability	Major	Pending

Description

The following is valid Opshin:

```python
1  a: Union[int, bytes] = 10
2  l = [a, 10, b'abcd']
```

`l` in this snippet will have inferred type `List[Union[int, bytes]]`. However, because in `AggressiveTypeInferencer.visit_List()`, the first list entry is used as the inferred item type, changing the order of these items will lead to compiler error, for example the following snippet will fail to compile:

```python
1  a: Union[int, bytes] = 10
2  l = [10, a, b'abcd']
```

Similarly, `AggressiveTypeInferencer.visit_Dict()` will use the type of the first key and the first value for the inferred type.

Recommendation

Find the most generic type contained in the `list` or `dict`, instead of using the first item type to determine the list or dict type.

Resolution

Pending

ID-U201 Fix Error Message in `visit_comprehension` of ATI

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

The error message on line 1185 of `opshin/type_inference.py` states "Type deconstruction in for loops is not supported yet." However, such for-loop-specific deconstructions should be acceptable as they were rewritten earlier in the pipeline.

Recommendation

Change error message to "Type deconstruction in comprehensions is not supported yet".

Resolution

Pending

ID-U202 Incorrect Hint when Using `Dict[int, int]` inside `Union`

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

When using `Dict[int, int]` inside a `Union` the following error is thrown: "Only Dict[Anything, Anything] or Dict is supported in Unions. Received Dict[int, int]".

When subsequently following the hint, and using `Dict` directly (without brackets), another error is thrown: "Variable Dict not initialized at access".

When using `List` in a similar way, a similarly incorrect hint is given.

Recommendation

Remove `Dict` and `List` from the hints. Also, improve the error message when using `Dict` and `List` inside `Union`.

Resolution

Pending

ID-U203 Incorrect Hints when Using `opshin eval` incorrectly

	Level	**Category**	**Severity**	**Status**
🟨	2	Usability	Minor	Pending

Description

When trying to evaluate a simple OpShin expression (e.g. `1 + 1`) defined in a file `example.py` using `opshin eval`, the following error is thrown: "Contract has no function called 'validator'. Make sure the compiled contract contains one function called 'validator' or eval using `opshin eval lib example.py`".

When subsequently trying the `opshin eval lib` command, the following error is thrown: "Libraries must have dead code removal disabled (-fno-remove-dead-code)".

When trying with `opshin eval lib -fno-remove-dead-code`, the following error is thrown: "Can not evaluate a library".

Recommendation

Remove the "or eval using opshin eval lib example.py" part of the first hint.

Resolution

Pending

ID-U204 Improve Error Message while Using List/Dict as function argument

	Level	**Category**	**Severity**	**Status**
🟨	2	Usability	Minor	Pending

Description

Newcomers to OpShin might try the following syntax:

```python
1  from opshin.prelude import *
2
3  def validator(_: List) -> None:
4      pass
```

This fails to compile, throwing the following error message:

`Variable List not initialized at access`. This error message doesn't help the user resolve the issue (`List[Anything]` must be used instead of `List`).

A similarly unhelpful error is thrown for `Dict`:

```python
1  from opshin.prelude import *
2
3  def validator(_: Dict) -> None:
4      pass
```

Recommendation

Either infer the types of `List` and `Dict` annotations as `List[Anything]` and `Dict[Anything, Anything]` respectively, or improve the error message by explaining the actual issue and providing a hint on how to resolve it.

Resolution

Pending

ID-U205 Non User-friendly Error when Using `Union` of single type

	Level	Category	Severity	Status
   	2	Usability	Minor	Pending

Description

In the following example validator, an argument is annotated with a `Union` of a single type:

```python
1  from opshin.prelude import *
2
3  @dataclass()
4  class A(PlutusData):
5      x: int
6
7  def validator(a: Union[A]) -> None:
8      assert isinstance(a, A)
```

An error is expected, but the compiler throws the following unrelated error message:

`'Name' object has no attribute 'elts'`.

Recommendation

The compiler should detect `Union`s containing only a single entry, and throw an explicit error.

Resolution

Pending

ID-U206 Inconsistent Treatment of Duplicate Entries in `Union`

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

Duplicate entries in `Union`s cause compiler errors, but duplicate entries in nested `Union`s do not.

Consider the following example validator:

```python
1  from opshin.prelude import *
2
3  @dataclass()
4  class A(PlutusData):
5      x: int
6
7  @dataclass()
8  class B(PlutusData):
9      x: bytes
10
11 def validator(a: Union[A, A, B]) -> None:
12     assert isinstance(a, A)
```

Expectedly, the compiler throws the following error:

`Duplicate constr_ids for records in Union: {'A': 1, 'B': 2}`.

But the following example validator compiles without errors:

```python
1  from opshin.prelude import *
2
3  @dataclass()
4  class A(PlutusData):
5      x: int
6
7  @dataclass()
8  class B(PlutusData):
9      x: bytes
10
11 def validator(a: Union[A, Union[A, B]]) -> None:
12     assert isinstance(a, A)
```

Recommendation

Flatten `Union`s before detecting duplicate entries. This will make `Union`s more user-friendly, especially when type aliases in deep transient imports are used, which might lead to unexpected duplication.

Optionally, a compiler step can be added to detect duplication of unresolved names at a single level within a `Union`, which might indicate a user mistake.

Resolution

Pending

ID-U207 Improve Documentation for Empty Literal `dicts`

Level	Category	Severity	Status
▉	2	Usability	Minor	Pending

Description

The type of an empty literal dict is never inferred, and as a consequence can only be used on the right-hand-side of an annotated assignment.

Consider the following example validator:

```python
1  from opshin.prelude import *
2
3  def my_len_fn(d: Dict[Anything, Anything]) -> int:
4      return len(d)
5
6  def validator(_: None) -> None:
7      assert my_len_fn({}) == 0
```

Compiling this example throws the following non user-friendly error: `list index out of range`. The same error is thrown when empty literal dicts are used in other expressions, for example in annotation-less assignments:

```python
1  def validator(_: None) -> None:
2      d = {}
3      pass
```

Recommendation

Add a note to the OpShin documentation that empty literal dicts must be assigned to a variable with type annotation before being usable (similar to the note already present about empty literal lists).

Resolution

Pending

ID-U208 `eval_uplc` Fails to Handle ComputationResult Errors

Level	Category	Severity	Status
2	Usability	Minor	Pending

Description

Evaluating an OpShin validator script using the `eval_uplc` command doesn't display runtime errors correctly. For example, calling the `eval_uplc` command with the example validator

```python
1  def validator(a: List[int]) -> None:
2      b = [x for x in a if x]
3      pass
```

gives the following output:

```
1   Starting execution
2   ------------------
3   Execution succeeded
4   Traceback (most recent call last):
5       File   "/home/user/.cache/pypoetry/virtualenvs/opshin-Gqoty4Xw-py3.9/bin/
    opshin", line 6, in <module>
6      sys.exit(main())
7    File "/home/user/Src/Opshin/opshin/opshin/__main__.py", line 518, in main
8      perform_command(args)
9       File   "/home/user/Src/Opshin/opshin/opshin/__main__.py",  line  416,  in
    perform_command
10     ret = uplc.dumps(ret.result)
11      File   "/home/user/.cache/pypoetry/virtualenvs/opshin-Gqoty4Xw-py3.9/lib/
    python3.9/site-packages/uplc/tools.py", line 105, in dumps
12     return u.dumps(dialect)
13  AttributeError: 'AssertionError' object has no attribute 'dumps'
```

Recommendation

In file `opshin/__main__.py`, in the last branch of `perform_command()`, test if `ret.result` is an error, and show an appropriate failure message in the case that it is.

Resolution

Pending

ID-U209 Fix Dict Access with Reordered `Union` Keys

Level	Category	Severity	Status
2	Usability	Minor	Pending

Description

Consider the following validator:

```python
1  from opshin.prelude import *
2
3  def validator(d: Dict[Union[int, bytes], int]) -> int:
4      key: Union[bytes, int] = 0
5      return d[key]
```

Compiling this example throws the following error:

```
subscript must have dict key type InstanceType(typ=UnionType(typs=[IntegerType(),
ByteStringType()]])) but has type InstanceType(typ=UnionType(typs=[ByteStringType(),
IntegerType()]))
```

Recommendation

In `union_types()` in `type_inference.py` : sort Union entries in an unambiguous way.

Resolution

Pending

ID-U210 Non User-friendly Error while Omitting Class Method Return Type

Level	Category	Severity	Status
2	Usability	Minor	Pending

Description

Consider the following example validator:

```python
1   from opshin.prelude import *
2
3   @dataclass()
4   class MyClass(PlutusData):
5       def my_method(self):
6           pass
7
8   def validator(_: None) -> None:
9       c = MyClass()
10      c.my_method()
```

Compiling this example gives the following error:

```
Invalid Python, class name is undefined at this stage
```
.

The error message doesn't help the user understand what is wrong with the code.

Recommendation

Detect class methods missing return types and throw an explicit error.

Resolution

Pending

ID-U211 `eval_uplc` Ignores `print()`

	Level	**Category**	**Severity**	**Status**
🟨	2	Usability	Minor	Pending

Description

Messages printed when evaluating a validator using `eval_uplc` aren't displayed.

Optimization level doesn't seem to have any impact on this.

Recommendation

Show messages from `print()` calls when evaluating a validator.

Resolution

Pending

ID-U212 Can't Use Empty Literal Lists in Arbitrary Expressions

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

When a variable is first declared with a type annotation (e.g., `def validator(x: List[int])`) and later assigned an empty value (e.g., `x = []`), the type checker fails to infer the type from the initial annotation and throws an unhelpful error `IndexError: list index out of range`. This occurs because the type checker treats annotated assignments (`x: List[int] = []`) and regular assignments (`x = []`) differently

```python
from typing import List

def validator(x: List[int]) -> int:
    x = []  # throws `IndexError` instead of inferring `List[int]`
    return len(x)
```

Recommendation

- Improve Error Messaging:

  ‣ Replace the cryptic `IndexError` with a clear, actionable error.
  ‣ If the variable has a prior type annotation, suggest: "Variable 'x' was previously annotated as 'List[int]'".

- Leverage Prior Annotations in `visit_Assign`:

  ‣ Modify the type checker to check for existing type annotations on the target variable during `visit_Assign`.
  ‣ If the target has a known type (e.g., from a prior annotation or parameter type), use it to infer the type of value of the expression.

Resolution

Pending

ID-U213 Improving Error Clarity

	Level	Category	Severity	Status
	2	Usability	Minor	Pending

Description

While the `opshin eval` command provides a valuable tool for evaluating scripts in Python, its error reporting can be enhanced to provide more user-friendly and informative feedback. Currently, when incorrect arguments or mismatched types are provided, the error messages may not clearly indicate the source or nature of the problem. For example:

```python
1  def validator(datum: WithdrawDatum, redeemer: None, context:
   ScriptContext) -> None:
2      sig_present = datum.pubkeyhash in context.tx_info.signatories
3      assert (
4          sig_present
5      ), f"Required signature missing, expected {datum.pubkeyhash.hex()} but got
   {[s.hex() for s in context.tx_info.signatories]}"
```

When we try to evaluate the validator using the below inputs:

```
opshin eval spending examples/smart_contracts/gift.py "{\"constructor\": 0,
\"fields\":[
{\"bytes\": \"1e51fcdc14be9a148bb0aaec9197eb47c83776fb\"}]}" "None" scriptcontext
```

Error Encountered:

```
ValueError: Expected hexadecimal CBOR representation of plutus datum but could not
transform hex string to bytes
```

The error is caused by the second argument, where "None" is passed instead of a valid Plutus data object for Nothing.

Recommendation

We recommend implementing more specific error messages that pinpoint the problematic argument, indicate its position, and clearly state the expected type.

Additionally, echoing the provided input, and suggesting corrections, for detailed debugging information could significantly improve the user experience and reduce troubleshooting time.

Resolution

Pending

ID-U214 Improve Documentation on Optimization Levels

Level	Category	Severity	Status
    2	Usability	Minor	Pending

Description

Currently, there is no clear documentation detailing the different optimization levels and the specific constraints that are enabled with each level. Providing this information would benefit users of OpShin, as it would give them a better understanding of which optimization configuration to choose based on their requirement.

Recommendation

The idea behind different Optimization levels(O1,O2,O3) and how the *UPLC* differs with each optimization level can be clearly documented with simple examples.

Resolution

Pending

ID-U215 Effect of Optimization Levels on Build Output

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

When building compiled code, OpShin creates the artifacts based on the default optimization level O1, where the conditions set are `constant_folding=False` and `remove_dead_code=True` .

As a result, the output *UPLC* contains more information than necessary, and therefore, the generated *CBOR* is also larger. This might increase the script size and makes debugging harder when used in off-chain transactions.

Recommendation

When building compiled code, OpShin could use the most aggressive optimizer, O3, as the default optimization configuration. This would allow users to directly utilize the optimized code without needing to specify any optimization levels during the build process.

Resolution

Pending

ID-U216 Improve Error for Out-of-Range Tuple Index

Level	Category	Severity	Status
🟨 2	Usability	Minor	Pending

Description

In `PlutoCompiler.visit_Subscript()` in `compiler.py`, a non user-friendly error is thrown if an out-of-range literal index used when accessing elements of a tuple.

Recommendation

Check out-of-range tuple indexing in `PlutoCompiler.visit_Subscript()` in order to throw a user-friendly error, instead of relying on the error thrown by the Pluthon codebase.

Resolution

Pending

ID-U217 Fix Misleading `opshin eval` Error Message

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

While evaluating the validator:

```python
1  def validator(x: bool) -> int:
2      return int(x)
```

using the command:

```
opshin eval any opshin/type_check.py '{"int":1}'
```

the following error occurs:

```
int() argument must be a string, a bytes-like object or a real number, not 'NoneType'.
```

After passing the argument according to the documentation, it says it's a not `NoneType` which means it is `None`. This error misleads about `eval` and also do not perform the Python evaluation of the script which `eval` is supposed to do.

Recommendation

From the documentation and the CLI-help, it is unclear how `eval` should be invoked.

The developer experience can be improved by removing the ability to call `eval` directly, and instead merging it with `eval_uplc`. `eval_uplc` would then perform a two-phase validation process.

In the first phase, the compiler should check whether the code is a subset of Python by running `eval`. If this phase fails, it should throw an error indicating that the code is not a valid subset. If it succeeds, the second phase can proceed by running `eval_uplc`, which converts the arguments into PlutusData and performs the validation.

Also, the result of `eval` and `eval_uplc` can be compared to ensure the *UPLC* program performs exactly as the developer intends.

Resolution

Pending

ID-U218 Inability to Assign to List Elements in Validator Functions

	Level	Category	Severity	Status
🟨	2	Usability	Minor	Pending

Description

In the provided code, the validator function attempts to modify an element of a list (x[0] += 1). However, the compiler raises an error:

`Can only assign to variable names, no type deconstruction`.

This restriction prevents list element assignment, which is a common and valid operation in Python and can be useful for on-chain code logic.

```python
1  def validator(x:List[int]) -> int:
2      x =[1,2,3,4]
3      x[0] += 1
4      return x
```

Recommendation

1. Extend the compiler to support assignments to list elements.
2. If supporting list element assignment is not feasible,enhance the error message to explain the limitation and suggest possible workarounds.

Resolution
Pending

ID-U101 Attaching File Name to Title in '.json' file

Level	Category	Severity	Status
1	Usability	Informational	Pending

Description

At present, the `opshin build` command compiles the validator, creates a target "build" directory and writes the artifacts to the build folder under the file name. The `blueprint.json` file is created, containing the compiled code, datum, and redeemer details. However, the field `title` in the blueprint.json file will always remain as "validator" as being assigned in the code. Suppose there is a function with name other than "validator", and when it is compiled using `opshin build lib` as expected by the OpShin language, the build artifacts will still have the title as "Validator" instead of the function name.

Recommendation

Although the file `blueprint.json` is primarily used for off-chain coding purposes, adding the validator's file name or function name along with the keyword 'Validator' as a title (e.g., Validator/ assert_sum) would be helpful for debugging and referencing during off-chain validation.

Resolution

Pending

ID-U102 Pretty Print Generated *UPLC* and *Pluthon*

	Level	**Category**	**Severity**	**Status**
	1	Usability	Informational	Pending

Description

When the OpShin code is compiled to *UPLC* using the `opshin eval_uplc` or `opshin compile` commands, the generated *UPLC* code is not formatted in a 'pretty-printed' form. Similarly, when compiled to *Pluthon* using the `opshin compile_pluto` command, the resulting code is also not presented in a 'pretty-printed' format. Instead, it is output directly to the terminal in a compact, unformatted style. This lack of formatting makes it more challenging to analyze or debug the resulting *UPLC* code, as the structure and readability of the code are compromised, which can hinder examination.

Recommendation

To improve the development experience, it would be beneficial to implement a method or tool that formats the *UPLC* output and *Pluthon* output and dumps it into a folder for each validator for easier interpretation and review.

Resolution

Pending

ID-U103 Determinisim of Constructor Ids

	Level	Category	Severity	Status
■	1	Usability	Informational	Pending

Description

```python
1  @dataclass
2  class DatumOne(PlutusData):
3      CONSTR_ID = 0
4      inttype: int
5
6  @dataclass
7  class DatumTwo(PlutusData):
8      CONSTR_ID = 1
9      inttype: bytes
```

If `CONSTR_ID` values are not explicitly defined for PlutusData classes, they are deterministically generated based on the class structure (e.g., field names, types, and class name) and when the classes are serialized to *UPLC*, constructor IDs are assigned automatically.

Recommendation

The current behavior of throwing an assertion error for duplicate `CONSTR_ID` values could be expanded to include a warning if no `CONSTR_ID` is provided, to alert developers about relying on automatically generated IDs.

Resolution

Pending

ID-U104 Method `to_cbor_hex()` Not Working

	Level	Category	Severity	Status
	1	Usability	Informational	Pending

Description

Though `to_cbor_hex()` is defined in the file `serialisation.py`, usage of the same throws an `TypeInferenceError`.

```python
@dataclass()
class Employee(PlutusData):
    name: bytes
    age: int

employee = Employee(b"Alice", 30)

def validator():
    print(employee.to_cbor_hex())
```

```
TypeInferenceError: Type Employee_0 does not have attribute to_cbor_hex
```

Recommendation

Inspect the code and include the method `to_cbor_hex()`.

Resolution

Pending

ID-U105 Nested Lists Not Handled Correctly

Level	Category	Severity	Status
1	Usability	Informational	Pending

Description

The following program:

```python
1  from typing import Dict, List, Union
2
3  def validator()-> List[List[int]]:
4      empty_List : List[List[int]] = [[]]
5      return empty_List
```

throws an error:

```
    empty_List : List[List[int]] = [[]]
                                    ^
IndexError: list index out of range
Note that opshin errors may be overly restrictive as they aim to prevent code with
unintended consequences.
```

It fails for empty nested lists like [[]],[[],[]] likely due to issues with type inference or no support for handling of nested structures.

Recommendation

Update the type inference system to handle nested empty lists (e.g., [[]], [[], []]) in assignments and returns.

Resolution

Pending

ID-U106 Error Messages Are Not Descriptive in Rewrite Transformers

	Level	Category	Severity	Status
	1	Usability	Informational	Pending

Description

In most of the rewrite transformers the error message for assertions are generic and do not provide enough context to help users understand the issue. For example, in the file

`rewrite/rewrite_import_dataclasses.py` the error message

`The program must contain one from dataclasses import dataclass` is repeated for various cases, making it difficult to diagnose specific problems.

Example:

```python
1  from pycardano import Datum as Anything, PlutusData
2  from dataclasses import dataclass as dc
3  @dc
4  class MyClass(PlutusData):
5      pass
6
7  def validator() :
8      return None
```

The issue here is the use of an alias name. The error message below does not convey the root cause of the problem properly.

```error
1  from dataclasses import dataclass as dc
2  AssertionError: The program must contain one 'from dataclasses import dataclass'
3  Note that opshin errors may be overly restrictive as they aim to prevent code
   with unintended consequences.
```

Recommendation

1. Improve error messages to be more specific. For example in this case if alias name is used, an error message could be something like this: "Aliasing 'dataclass' is not allowed. Use 'from dataclasses import dataclass' directly."

2. Review all assertion error messages in the following transformer rewrites.

- rewrite_import_dataclasses.py
- rewrite_import_typing.py

Resolution

Pending

ID-U107 Unclear Error for Unimplemented Bitwise XOR

	Level	Category	Severity	Status
▉	1	Usability	Informational	Pending

Description

When using unsupported operators (e.g., bitwise XOR ^) in operations, the evaluation throws a `RecursionError: maximum recursion depth exceeded` instead of a clear error indicating the operator is unimplemented. However this compiles when the optimization of constant expressions is turned on.

```python
1
2 def validator():
3     x = hex(1 ^ 256)  # Throws `RecursionError` instead of "Operator '^' not supported"
4     print(x)
```

Recommendation

- Detect unsupported operators during parsing/compilation and raise a descriptive error (e.g., CompilerError: Operator '^' (bitwise XOR) is not supported).

- Include a list of supported operators in the error message (e.g., Supported operators: +, -, *, /, &, |).

- Prioritize implementing commonly used missing operators or explicitly document unsupported ones.

Resolution

Pending